Titre: Title:	Conception d'un outil de modélisation intégré pour l'indexation et l'analyse de trace
Auteur: Author:	Simon Delisle
Date:	2015
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Delisle, S. (2015). Conception d'un outil de modélisation intégré pour l'indexation et l'analyse de trace [Master's thesis, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/2035/

UNIVERSITÉ DE MONTRÉAL

CONCEPTION D'UN OUTIL DE MODÉLISATION INTÉGRÉ POUR L'INDEXATION ET L'ANALYSE DE TRACE

SIMON DELISLE
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
DÉCEMBRE 2015

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

CONCEPTION D'UN OUTIL DE MODÉLISATION INTÉGRÉ POUR L'INDEXATION ET L'ANALYSE DE TRACE

présenté par : DELISLE Simon
en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de :

M. BOIS Guy, Ph. D, président
M. DAGENAIS Michel, Ph. D., membre et directeur de recherche
M. GUIBAULT François, Ph. D., membre

DÉDICACE

À mon père,
qui m'a tant appris
et qui m'a toujours supporté.

REMERCIEMENTS

Tout d'abord, je tiens à remercier mon directeur de recherche, Michel Dagenais, pour le support, la guidance et la confiance qu'il a su m'apporter tout au long de mes recherches. Son soutien et sa disponibilité m'ont permis de passer au travers des différents obstacles rencontrés et il a su me remettre sur le bon chemin lorsqu'il était nécessaire de le faire. Je remercie Michel de m'avoir donné l'opportunité d'accomplir cette maîtrise. J'ai pu en apprendre davantage sur différents sujets et cela m'a permis d'avancer.

Je souhaite remercier les partenaires du laboratoire pour leur soutien financier dans ce projet de recherche. Au-delà du financement, les employés de chez Ericsson étaient présents et prêts à m'aider dans les différentes étapes de mon travail. Je veux aussi remercier tous mes collègues du laboratoire DORSAL sans qui cette ambiance de travail n'aurait pas été aussi agréable. Merci à Geneviève, Naser et Francis de m'avoir guidé tout au long de mes travaux de recherche.

J'aimerais remercier ma famille, ma copine et mes amis pour leur support constant. Sans eux, cette maîtrise n'aurait pas été possible. Merci à François, Mathieu et Raphaël de m'avoir enduré dans ce petit local où nous avons forgé de bons souvenirs.

Enfin, merci aux membres du jury et aux autres lecteurs de prendre le temps de lire ce présent mémoire.

RÉSUMÉ

Avec une diversification des systèmes informatiques, les développeurs font face à des problèmes de performance de plus en plus complexes à identifier. Les techniques conventionnelles, comme le débogage, s'avèrent de moins en moins efficaces dans des systèmes distribués ou fortement multi-coeur. Les traces d'exécution deviennent une solution intéressante afin de comprendre le comportement d'un système ou d'une application.

Des traceurs comme *Linux Trace Toolkit next generation* et des outils d'analyse comme Trace Compass ont vu le jour afin d'aider les développeurs tant au niveau du noyau Linux qu'au niveau des applications en espace utilisateur.

Les traces contiennent beaucoup d'informations qui peuvent être parfois inutiles d'exposer à l'utilisateur. Dans des situations où la quantité d'événements capturés lors du traçage est élevée, l'utilisateur peut difficilement analyser et retrouver l'information qui lui est réellement utile seulement en regardant tous ces événements. C'est pourquoi nous avons besoin d'un outil qui analysera et affichera les bonnes informations au développeur. Par conséquent, l'analyse constitue un défi de conception et doit être efficace afin d'extraire l'information pertinente sous différents angles. Trace Compass utilise une analyse basée sur une machine à état afin de montrer les différents états des fils d'exécution en fonction du temps.

Par contre, cette technique est spécifique pour chaque type de trace et on doit programmer, en Java, nous-mêmes une nouvelle analyse pour chaque type de trace. Des travaux effectués précédemment dans le laboratoire DORSAL à l'École Polytechnique de Montréal ont permis de rendre ce processus de création d'analyses plus flexible. Un langage déclaratif écrit dans un fichier Extensible Markup Language (XML) fut introduit. L'idée était de remplacer le fournisseur d'état, module permettant de convertir les événements en états actuellement écrit en Java, par une méthode dirigée par les données.

Malgré le fait que cette solution améliore la flexibilité des analyses, elle n'est pas simple d'utilisation. En effet, il faut sortir de notre environnement de traçage pour écrire un fichier qui peut rapidement devenir volumineux. De plus, l'ajout de fonctionnalités rendrait le tout plus complexe à gérer et à créer.

Afin d'amener la création d'analyses encore plus loin, nous avons créé un outil de modélisation qui permet de capturer de façon conviviale toutes les informations se rattachant à l'analyse de trace et ayant pour objectif de remplacer l'écriture du XML. Pour y arriver, nous utiliserons un modèle de machine à état pour représenter l'analyse. À l'aide des outils fournis dans

Eclipse, il sera possible de construire un outil qui répondra aux critères. Nous avons utilisé avec succès notre outil pour modéliser des analyses complexes et nous avons vérifié que ces analyses générées par notre outil donnent le même résultat que celles effectuées en XML.

Bref, les résultats montrent qu'il est possible d'utiliser la modélisation afin de capturer tous les éléments nécessaires pour bâtir une analyse. De plus, les extensions possibles apporteront un plus à l'utilisation de la modélisation dans le domaine du traçage.

ABSTRACT

Developers need advanced tools to identify performance problems in increasingly complex computer systems. In particular, abnormal execution latency in multi-core or distributed systems cannot be diagnosed using traditional interactive debuggers. An effective technique consists in gathering execution traces, including the timing of events, to effectively understand the execution of the application and the system as a whole. The Linux Trace Toolkit next generation tracer and the Trace Compass analysis tool are key components to achieve a better comprehension of a system, both at the kernel and user space levels.

Analyzing manually a raw trace is a difficult task for the user, especially when the system produces a high rate of events in parallel at multiple levels. The main task is to recover the system state based on the event stream, computing metrics of resource usage, and a higer-level representation of the execution behavior. To reduce the cognitive load, powerful analysis tools are required to provide meaningful representations of the trace. The challenge is to extract the relevant information from different perspectives and display it in an intuitive and compact manner.

For instance, Trace Compass provides an interactive time view of processes on a computer and their state, that allows to inspect the system at any time during its execution. However, the analysis is specific to a kernel trace. Creating a new analysis, either for a different trace type or observing other aspects of the system, may be done by writing code. More recently, an abstraction of state-based analysis from a trace was realized in the DORSAL group at Polytechnique Montreal. This technique relies on a declarative language to describe the state machine, and a generic engine to recover the state according to the incoming events. The analysis is written in an XML file, and no coding is required, which reduces the burden of creating new state-based analysis.

Despite the fact that declarative analysis reduces the difficulty and the time to create new analysis, editing an XML file remains difficult. The user needs to learn the set of XML tags, and create a mental representation of the state machine textual description, that may have numerous states and transitions, which can easily lead to mistakes.

Therefore, our goal is to replace direct XML editing by a graphical editor, specifically to represent state machine analysis in Trace Compass. This modeling tool would simplify the design of new analyses and prevent errors in the models. Our modeling tool implementation, based on the Eclipse framework, includes every aspect of state-based analysis design. We used successfully our tool to realize complex analyses, and verified that they produce the same

result as their XML counterpart. Our results show that our modeling tools offers practical benefits over a generic XML editor in terms of ease of use, safety and functionality.

TABLE DES MATIÈRES

DÉDICACE iii

REMERCIEMENTS iv

RÉSUMÉ v

ABSTRACT vii

TABLE DES MATIÈRES ix

LISTE DES TABLEAUX xii

LISTE DES FIGURES xiii

LISTE DES SIGLES ET ABRÉVIATIONS xiv

CHAPITRE 1 INTRODUCTION 1
1.1 Définitions et concepts de base 1
1.1.1 Événement 1
1.1.2 Attribut 1
1.1.3 Analyse 2
1.1.4 Modélisation 2
1.1.5 Développement dirigé par les modèles 3
1.2 Éléments de la problématique 4
1.3 Objectifs de recherche 4
1.4 Plan du mémoire 5

CHAPITRE 2 REVUE CRITIQUE DE LA LITTÉRATURE 6
2.1 Le traçage 6
2.1.1 Définition générale 6
2.1.2 Traçage noyau 7
2.1.3 Traçage en espace utilisateur 8
2.1.4 Traçage sous Windows 9
2.2 L'analyse de trace 9
2.2.1 Gestionnaire d'état 9
2.2.2 Stockage de l'état dans une structure dédiée 10

2.2.3 Stocker l'état dans la trace elle-même 11
2.2.4 Analyse de trace en ligne 11
2.2.5 Visualisation de trace 12
2.3 Langages de modélisation 13
2.3.1 Langages déclaratifs 14
2.3.2 Langages impératifs 14
2.3.3 Langages basés sur les automates 15
2.3.4 DSL - Langages spécifiques au domaine 16
2.4 Modélisation 17
2.4.1 Outils de modélisation 17
2.4.2 UML 19

CHAPITRE 3 MÉTHODOLOGIE 22
3.1 Rappel de la problématique 22
3.2 Démarche 22

CHAPITRE 4 ARTICLE 1 : INTEGRATED MODELING TOOL FOR INDEXING AND ANALYZING STATE MACHINE TRACE 24
4.1 Abstract 24
4.2 Introduction 25
4.3 Related Work 26
4.3.1 Tracing 26
4.3.2 Trace analysis 28
4.3.3 Languages 29
4.3.4 Modeling 30
4.4 Architecture 31
4.5 Model Specification 32
4.5.1 Modeling tool 32
4.5.2 State machine 33
4.5.3 State 33
4.5.4 Transition 34
4.5.5 Condition 34
4.6 Application and Results 37
4.7 Conclusion and Future Work 44
4.8 Acknowledgment 45

CHAPITRE 5 DISCUSSION GÉNÉRALE 46

5.1 Retour sur les objectifs et la méthodologie 46
5.2 Intégration de nouvelle fonctionnalités 47
5.2.1 Filtres 47
5.2.2 Événement synthétique 48
5.2.3 Ajout à l'outil de modélisation 48
5.3 Utilisation de l'outil comme une vue 50
5.4 Gestion des fichiers et préférences 50
5.4.1 Nouveaux fichiers 51
5.4.2 Interaction entre les fichiers 51
5.4.3 Architecture possible 51

CHAPITRE 6 CONCLUSION ET RECOMMANDATIONS 55
6.1 Synthèse des travaux 55
6.2 Limitations de la solution proposée 56
6.3 Améliorations futures 56

RÉFÉRENCES 58

LISTE DES TABLEAUX

Table 4.1 Number of attributes for different analysis 43
Table 4.2 Number of element in the model for different analysis 43

LISTE DES FIGURES

Figure 1.1	Exemple d'un arbre d'attributs	2
Figure 1.2	Analyse Trace Compass	3
Figure 2.1	Conversion d'événements en états	10
Figure 2.2	Vues du logiciel Trace Compass	13
Figure 2.3	Diagramme illustrant la structure logique de la machine à état en UML	20
Figure 4.1	Traditional trace analysis approach.	31
Figure 4.2	Trace analysis with XML specification.	31
Figure 4.3	Trace analysis with the proposed modeling tool.	32
Figure 4.4	Example of an attribute tree	33
Figure 4.5	State machine and state that will be used for the Linux kernel analysis	34
Figure 4.6	Example of transitions that will be used for the Linux kernel analysis	35
Figure 4.7	Example of condition	36
Figure 4.8	Creation of attributes based condition	36
Figure 4.9	Build one condition using single conditions	36
Figure 4.10	The creation of an attribute tree	37
Figure 4.11	Linux kernel attribute tree	38
Figure 4.12	State machine of the Linux kernel analysis	39
Figure 4.13	CPU state machine used in the Linux kernel analysis	39
Figure 4.14	Request analysis attribute tree	40
Figure 4.15	State machine of Trace Compass requests analysis	41
Figure 4.16	Example of one condition used in the requests analysis	41
Figure 4.17	Definition of state color for views generation	42
Figure 4.18	View generated from the request analysis	42
Figure 5.1	Exemple de vue utilisée dans Papyrus	50
Figure 5.2	Dépendances entres les différents fichiers	52
Figure 5.3	Exemple de hiérarchie pour les préférences de Trace Compass	53

LISTE DES SIGLES ET ABRÉVIATIONS

LTTng	Linux Trace Toolkit next generation
CTF	Common Trace Format
UST	User-Space Tracer
ETW	Event Tracing for Windows
WPT	Windows Performance Toolkit
DSL	Domain Specific Language
IDS	Intrusion Detection System
EMF	Eclipse Modeling Framework
GMF	Graphical Modeling Framework
UML	Unified Modeling Language
OMG	Object Management Group
XML	Extensible Markup Language

CHAPITRE 1 INTRODUCTION

L'utilisation de techniques conventionnelles, comme le débogage, peut s'avérer inefficace pour trouver les problèmes de performance présents dans les systèmes distribués et multi-coeur. Les traces d'exécutions, ayant un impact minimal sur le système, fournissent une solution adéquate à ce problème. Les traces procurent des données détaillées sur le système tracé et pourront servir à effectuer des analyses plus poussées. En effet, les données brutes seules ne sont pas d'une grande utilité, puisque la trace peut contenir beaucoup d'informations. Parmi ces informations, on retrouve tous les événements qui se sont produits dans un intervalle de temps, ainsi que les details associés aux événements capturés. L'utilisation de logiciels d'analyse, comme Trace Compass, Jumpshot ou les outils fournis par Microsoft, devient donc nécessaire. Dans ce mémoire, Trace Compass sera utilisé pour effectuer les tests et valider la solution.

Par contre, une limitation majeure de ces outils est la flexibilité. Ce qui a pour effet de restreindre l'utilisateur à se servir des analyses préconçues présentes dans les logiciels. Étant donné que chaque problème est particulier, les développeurs ont besoin de concevoir facilement leurs analyses. Pour ce faire, nous présenterons un outil de modélisation simple d'utilisation qui, combiné au traçage, permettra de définir et d'effectuer des analyses détaillées.

1.1 Définitions et concepts de base

1.1.1 Événement

Dans le contexte d'une trace, un événement représente une action qui a lieu sur le système ou dans l'application tracée. Un événement est déclenché lorsqu'une instruction instrumentée est atteinte. Un événement est ponctuel et ne possède pas de durée.

L'événement est constitué de trois éléments. Tout d'abord, une estampille de temps à laquelle il est survenu. Puis, le type correspondant à l'événement, par exemple un appel système ou encore une entrée ou une sortie de fonction. Finalement, l'événement contient de l'information additionnelle sur le contexte.

1.1.2 Attribut

Afin de pouvoir représenter l'état complet d'un système, il faut le diviser en ressources. Un attribut est la plus petite subdivision de ce modèle. Il contient une unique valeur d'état et

permettra de représenter un système complexe. Par exemple, les valeurs peuvent être l'état d'un processeur, un numéro de processus ou encore l'état d'un fichier utilisé par le système. Lorsqu'un attribut est créé, il doit toujours posséder une valeur, qui peut être nulle, afin de s'assurer de son existence en tout temps.

Afin d'obtenir l'état du système, les attributs sont organisés sous forme d'arbre. La Figure 1.1 est un exemple de l'organisation de cet arbre. Chaque attribut est accessible par un chemin unique à partir de la racine.

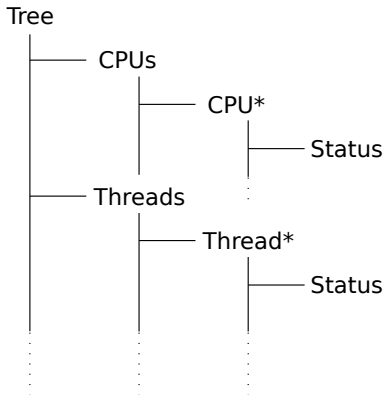

Figure 1.1 Exemple d'un arbre d'attributs

1.1.3 Analyse

L'analyse de traces permet de faire le lien entre les données brutes et l'information que l'utilisateur veut obtenir en rapport à son problème. Les analyses sont souvent faites *a posteriori* afin de pouvoir produire un résultat plus précis en limitant l'impact sur le système tracé. Les analyses peuvent être optimisées indépendamment de la trace.

Les vues utilisent alors le modèle qui a été généré à partir des événements de la trace. L'exemple d'analyse de la Figure 1.2 montre les différents états des processus. Cette analyse utilise un gestionnaire qui permet de convertir les événements en états, le tout sera discuté plus tard dans ce travail.

1.1.4 Modélisation

La modélisation de logiciels permet d'exprimer de l'information sur un savoir ou sur un système en utilisant une structure définie. Cette dernière est définie par des règles bien précises. Unified Modeling Language (UML) est un exemple de règle très répandu dans le domaine du logiciel.

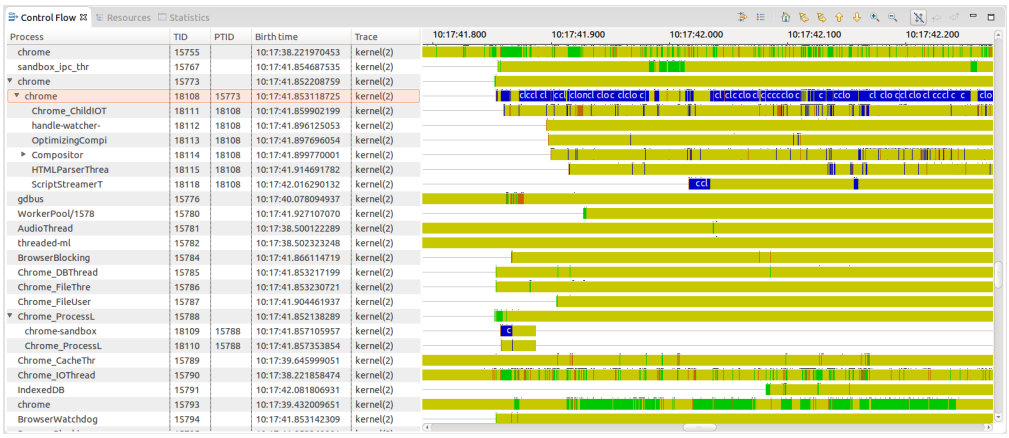

Figure 1.2 Analyse Trace Compass

Dans le cadre de ce mémoire, la modélisation graphique sera au coeur de nos travaux. Avec la modélisation graphique, il est possible de modéliser des processus de développement logiciel, des diagrammes de classe ou de composante, des modèles d'affaire et bien plus encore.

1.1.5 Développement dirigé par les modèles

Le développement dirigé par les modèles est une pratique en ingénierie qui permet de décrire, au travers de modèles, un problème ainsi que la solution. L'attention est portée sur une représentation abstraite d'un savoir dans un domaine d'application particulier.

Cette approche a pour but de favoriser la productivité du développement de produits logiciels. Par exemple, la réutilisation de patrons de conception permet de simplifier le processus de conception de logiciels. Par contre, ce paradigme est efficace seulement si le modèle est compréhensible par l'utilisateur qui est familier avec le domaine.

1.2 Éléments de la problématique

Comme mentionné précédemment, l'analyse de traces est un atout important lorsqu'on veut diagnostiquer certains types de problèmes de performance, comme la latence ou encore des délais intermittents. Ces problèmes deviennent de plus en plus difficiles à détecter. En effet, les développeurs créent leurs applications avec l'idée de les optimiser pour qu'elles soient plus performantes. Cette recherche de performance est causée par les nombreuses avancées dans les domaines de la virtualisation, l'informatique nuagique, les mobiles et les ordinateurs multiprocesseurs.

Les développeurs doivent donc recourir à des outils adaptés à leurs besoins pour surmonter ces nombreux défis. Le traçage en fait partie. Les traceurs d'aujourd'hui offrent une grande flexibilité et permettent de récolter toute l'information nécessaire. Par contre, ce n'est pas le cas des analyses qui sont souvent intégrées dans les logiciels d'analyse, et donc difficilement adaptables aux situations particulières. Dans les efforts pour les rendre plus flexibles, un fichier de spécification en XML fut introduit dans Trace Compass. Ce nouveau langage permettait de définir des analyses personnalisées. En contrepartie, cette solution nécessite l'apprentissage d'un nouveau langage et le fichier de spécification peut devenir rapidement volumineux, donc plus facile de s'y perdre et la possibilité d'y insérer des erreurs est plus grande.

Pour ces raisons, nous avons besoin d'un outil qui permettra de concevoir des analyses. De plus, cet outil devra être extensible afin d'y intégrer de futurs travaux comme la spécification de filtres ou d'événements synthétiques.

1.3 Objectifs de recherche

La problématique énoncée à la section précédente permet de formuler la question de recherche suivante : est-ce possible de capturer sous forme conviviale, typiquement dans un éditeur de machine à états finis, tous les éléments se rattachant à l'analyse de traces ?

À la suite de cela, nous pouvons définir les objectifs spécifiques suivants :

1. Développer un métamodèle de machine à états finis qui sera adapté aux besoins du traçage.
2. Définir l'arbre d'attributs relié au type de trace à analyser.
3. Spécifier l'état du système tracé à partir de l'outil de modélisation.
4. Valider que la solution conserve ou améliore les fonctionnalités déjà présentes.

Ces objectifs permettront de répondre à l'objectif principal de concevoir un outil de modélisation afin de construire simplement et facilement des analyses flexibles et complètes. Le premier objectif est de développer un modèle qui permettra, d'une part, d'être manipulé et instancié à l'aide d'une interface graphique et, d'autre part, de remplacer totalement l'écriture d'une analyse déclarative en XML. Nous allons tenter de remplacer les fonctionnalités présentes dans le XML par des éléments de machine à état.

Le deuxième objectif est de permettre à l'utilisateur de définir un arbre d'attributs à l'aide de l'interface graphique afin de faciliter la conception d'une analyse. En effet, les attributs ne changent pas pour un type de trace, seules les valeurs qu'ils contiennent changent. Il est donc utile d'avoir cet arbre pour éviter de saisir l'information encore et encore dans le diagramme de machine à état.

Les derniers objectifs permettront de vérifier si le modèle convient aux besoins. Ils permettront de voir s'il est possible de remplacer complètement l'écriture du fichier XML en bâtissant des analyses avec notre outil et en validant celles-ci avec les anciennes analyses faites en XML.

1.4 Plan du mémoire

Le mémoire est structuré de la façon suivante. Le chapitre 2 se veut une revue de littérature sur l'état de l'art dans les domaines du traçage, de l'analyse de traces, des langages et Domain Specific Language (DSL), puis de la modélisation. Dans le chapitre 3, la démarche utilisée est présentée. Le chapitre 4 contient l'article "ARTICLE 1 : INTEGRATED MODELING TOOL FOR INDEXING AND ANALYZING STATE MACHINE TRACE" qui présente les travaux de recherche effectués afin de répondre à la problématique fixée précédemment. Une discussion générale s'ensuit au chapitre 5. Elle contient des résultats complémentaires et des améliorations à la solution proposée. Finalement, le chapitre 6 termine ce mémoire avec les améliorations possibles ainsi que les travaux futurs.

CHAPITRE 2 REVUE CRITIQUE DE LA LITTÉRATURE

Dans ce chapitre, nous présenterons l'état de l'art dans le domaine du traçage, de la modélisation de systèmes et la modélisation graphique. Nous aborderons quatre thèmes en lien avec notre sujet. Les deux premiers sujets qui seront abordés porteront, respectivement, sur le traçage et sur l'analyse de traces. Ils permettront de comprendre la provenance, la génération et l'analyse de traces. Par la suite, l'état de l'art des langages de modélisation sera présenté. Il permettra de montrer les langages existants et leurs faiblesses. La modélisation graphique terminera ce chapitre. Dans cette partie, il sera question des différents outils de modélisation ainsi que leur architecture et leur fonctionnement.

2.1 Le traçage

Dans cette section, nous aurons l'occasion de présenter le travail qui a été fait dans le domaine du traçage, principalement sur le système d'exploitation Linux. L'ouverture de cette plateforme en fait un choix intéressant pour l'étude du traçage. Toutefois, le traçage sur Windows n'est pas exclu, car il est possible d'appliquer la solution à une trace Windows, mais seulement une brève introduction sera présentée dans cette section.

2.1.1 Définition générale

Le traçage consiste à récupérer de l'information sur une application ou sur un système lors de son exécution. L'utilisation du traçage et les données contenues dans la trace ont pour but de détecter les problèmes de performance. Afin d'obtenir un portrait représentatif de l'application ou du système, le traçage ne doit pas perturber l'exécution normale de ceux-ci. Le traçage, contrairement au débogage, n'interrompt pas l'exécution. L'analyse de ces traces sera vue en détail plus tard.

On peut obtenir une trace de notre programme en y insérant des points de trace dans celui-ci. Il est possible de le faire de façon statique ou dynamique. Dans le premier cas, les points de trace sont ajoutés manuellement à même le code en utilisant la macro TRACE_EVENT (Rostedt, 2010) dans le cas du noyau Linux. Il est donc nécessaire d'avoir accès au code source. Dans le deuxième cas, il est inutile de recompiler le programme pour insérer les points de trace. En effet, ces derniers sont ajoutés dynamiquement lors de l'exécution du programme. Kprobes (Goswami, 2005) est un mécanisme qui permet de réaliser cela pour le noyau Linux. Par contre, le surcoût qu'engendre cette méthode perturbe le programme.

2.1.2 Traçage noyau

SystemTap

Le traceur SystemTap a été développé en 2005 par Red Hat (Prasad et al., 2005). Il est destiné aux administrateurs système afin de faciliter l'accès au traçage du noyau. Pour y arriver, SystemTap utilise un langage de script qui permet d'utiliser la macro TRACE_EVENT pour l'instrumentation statique ainsi que l'utilisation de kprobes pour les cas où les points d'instrumentation statiques voulus ne sont pas disponibles. Le calcul des statistiques pendant le traçage et la flexibilité du traceur en font un choix intéressant lorsque vient le temps de diagnostiquer des problèmes de performance. Par contre, les traces peuvent devenir rapidement volumineuses. En effet, SystemTap ne possède pas de format de trace compact. Le tout est écrit dans un fichier texte ou directement dans une console. De plus, comme l'analyse des résultats se fait pendant le traçage, il est impossible de faire des analyses plus complexes ou personnalisées par la suite. Pour ces raisons, l'utilisation de SystemTap n'est pas applicable dans notre situation.

Perf

Perf permet de tracer le noyau Linux en utilisant les compteurs de performance matériel ou logiciel. Depuis 2009 (Edge, 2009), il fait partie des outils intégrés dans le noyau Linux. Par la suite, le support pour TRACE_EVENT ainsi que l'utilisation de kprobes ont été ajoutés. Perf fonctionne par échantillonnage de valeurs. Lorsqu'une certaine limite fixée est atteinte, une interruption est lancée et on enregistre les valeurs des compteurs. Les analyses obtenues sont donc sous forme de statistiques. Cette méthode par échantillonnage permet d'avoir un faible impact sur le système tracé. En contrepartie, les données recueillies sont moins précises.

DTrace

DTrace est le traceur de référence sur les systèmes Solaris. Il a été présenté en 2004 (Cantrill et al., 2004). Un livre a aussi été écrit par Gregg and Mauro (2011) expliquant le fonctionnement du langage (langage D) et l'utilisation de ce traceur. DTrace est reconnu pour n'avoir aucun impact sur le système tracé lorsque les points de trace sont désactivés. Pour ce faire, il utilise l'instrumentation dynamique du noyau ou des applications en espace utilisateur. DTrace possède son propre langage semblable au C pour décrire les actions à prendre lorsqu'un prérequis est respecté. Cela permet d'avoir une interface unifiée pour le traçage du noyau et d'application en espace utilisateur.

LTTng

Introduit en 2006 (Desnoyers and Dagenais, 2006), le traceur Linux Trace Toolkit next generation (LTTng) permet d'obtenir des traces du noyau Linux. Il a pour objectif de minimiser son impact sur le système tracé. La version 2.0 (Mathieu Desnoyers, 2012) est présentée sous forme de module noyau et utilise le format libre Common Trace Format (CTF) (Desnoyers, 2011) pour enregistrer la trace sur le disque. LTTng utilise la macro TRACE_EVENT présente dans le noyau Linux et kprobes pour l'instrumentation dynamique. Afin d'obtenir les meilleures performances possible, les événements recueillis sont d'abord écrits dans des tampons circulaires, puis consommés pour être écrits sur le disque. Ce traceur supporte aussi l'ajout d'informations sur le contexte à chaque événement. Ceci permet, par exemple, de connaître le processus qui est en cours d'exécution sur un processeur au moment de l'événement. De plus, les compteurs de performance, considérés comme de l'information de contexte, peuvent être enregistrés pour chaque événement.

2.1.3 Traçage en espace utilisateur

Nous avons vu que le traçage du noyau peut fournir beaucoup d'informations sur l'exécution d'un système ou d'une application. Parfois, nous voulons recueillir de l'information plus spécifique à notre application. Dans le même ordre d'idées que le traçage du noyau, le traçage en espace utilisateur permet de tracer une application en limitant l'impact sur la performance de celle-ci. C'est une alternative au débogage traditionnel comme les points d'arrêt ou `printf`.

Les traceurs DTrace et SystemTap ont une composante espace utilisateur. Par contre, l'impact sur la performance de l'application tracée est élevé en raison des appels système à chaque point de trace rencontré. Dans le cas de SystemTap, la composante qui permet le traçage en espace utilisateur est passée de utrace (Corbet, 2007) à uprobes (Jim Keniston, 2010), qui est semblable à kprobes.

LTTng-UST

LTTng-UST (Fournier et al., 2009) est la composante *User-Space Tracer* de LTTng présenté précédemment. La particularité de celui-ci est qu'il s'exécute entièrement en espace utilisateur. Il fonctionne, tout comme dans le cas du noyau, en utilisant des points de trace statique définis dans le code de l'application. Pour obtenir une performance optimale, LTTng-UST écrit directement les événements dans une mémoire partagée. Les événements en mémoire peuvent être consommés, sans blocage, par un processus compagnon qui se charge d'écrire le tout sur le disque. Comme mentionné dans (Fournier et al., 2009), cette façon de faire

est plus performante que DTrace. La trace en sortie est dans le format CTF mentionné précédemment.

2.1.4 Traçage sous Windows

Event Tracing for Windows (ETW) est la plateforme de traçage, conçue par Microsoft (Insung Park, 2007), pour les systèmes d'exploitation Windows et introduite avec Windows 2000. Elle permet de tracer tant au niveau du noyau qu'au niveau applicatif. Grâce à une interface de programmation, toutes les applications peuvent devenir des fournisseurs d'événements. ETW utilise un système semblable aux tampons dans LTTng pour afficher en direct ou dans un fichier de trace l'information recueillie. Une particularité intéressante de cette infrastructure est la prise de clichés de l'état du système avant le début de la session de traçage. En effet, cela nous permet d'avoir de l'information sur le contexte d'exécution et ainsi faciliter l'analyse *a posteriori*.

2.2 L'analyse de trace

Afin de détecter efficacement les problèmes de performance, il faut analyser les traces capturées précédemment. En séparant la partie du traçage et de l'analyse, on peut obtenir un traçage ayant le moins d'impact possible sur la performance du système tracé et on peut se permettre des analyses plus coûteuses, mais plus précises.

Il existe plusieurs méthodes d'analyse. Dans cette section, il sera question du gestionnaire d'état utilisé par Trace Compass [1] ainsi que la façon de sauvegarder cette information sur le disque. Puis, nous aborderons différentes méthodes d'analyses intégrées à même la trace. Finalement, nous regarderons brièvement l'analyse en ligne et la visualisation de traces.

2.2.1 Gestionnaire d'état

L'analyse d'une trace à l'aide de la méthode par gestionnaire d'état permet de modéliser l'état d'un système. Cette méthode consiste à bâtir une machine à état à partir des événements de la trace. La figure 2.1 montre la façon de transformer des événements en états. Cette machine est créée en ordre chronologique lors de la première lecture de la trace. Dans la section suivante, nous aborderons la façon de sauvegarder efficacement ces données. Cette méthode permet de faire des requêtes de manière efficace à n'importe quel endroit dans la trace afin d'obtenir l'état du système à un temps donné. Par contre, le gestionnaire d'état perd de la

1. `http://projects.eclipse.org/projects/tools.tracecompass`

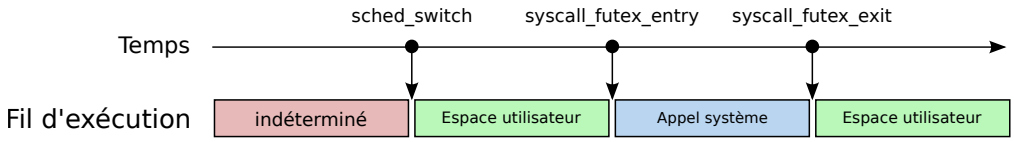

Figure 2.1 Conversion d'événements en états

précision lorsqu'il y a des événements perdus. En effet, un état peut être inexact lorsque l'événement perdu s'avère être la fin ou le début d'un état. Nous aurions alors un état qui en englobe deux autres. Dans le cas illustré à la figure 2.1, si l'événement `syscall_futex_entry` est perdu, alors l'état serait resté `Espace utilisateur` et nous n'aurions pas vu l'appel système. De plus, lors de l'initialisation de la machine, on ne connaît pas l'état de départ, donc nous sommes dans un état indéterminé. Pour y remédier, on peut ajouter de l'information sur le contexte précédant le début de la prise de la trace. Comme mentionné plus haut, ETW permet d'obtenir cette information tout comme LTTng en utilisant la fonction `lttng_statedump_process_state`.

Dans l'article de (Giraldeau et al., 2011), celui-ci utilise la méthode du gestionnaire d'état pour extraire différentes métriques (utilisation du CPU, utilisation de mémoire, etc.) du système tracé.

2.2.2 Stockage de l'état dans une structure dédiée

Un des problèmes de la méthode du gestionnaire d'état est que si l'on construit la machine complète en mémoire, on est rapidement limité par la taille de la trace que l'on peut analyser. Il existe une méthode pour réduire la taille de la machine en sauvegardant des clichés à intervalles réguliers pour ensuite reconstruire la machine plus précise en partant d'un de ces points lorsqu'on fait une requête à un temps donné. Par contre, nous sommes encore limités sur la taille de la trace dans le sens que la performance des requêtes en est affectée.

La solution présentée par Montplaisir-Gonçalves et al. (2013) permet de remédier à ce problème en proposant une structure en arbre pour sauvegarder le tout sur disque. Cet arbre de recherche permet de faire des requêtes en temps logarithmique.

Cette solution a pour hypothèse que les intervalles arrivent en ordre croissant de temps de fin. Il n'y a donc pas de retour dans le passé. Ce critère permet d'avoir un arbre équilibré sans avoir à le rebalancer.

On peut utiliser cette structure afin de gérer et de visualiser les valeurs des états d'un système (Montplaisir et al., 2013). Les valeurs d'état sont modélisées comme des intervalles et

sauvegardées dans l'arbre. De plus, l'introduction de l'arbre d'attributs sera fort utile pour la solution proposée dans ce mémoire.

2.2.3 Stocker l'état dans la trace elle-même

On peut aussi sauvegarder les états dans la trace. Cette méthode permet de faciliter la visualisation de traces dans des logiciels comme Jumpshot (Zaki et al., 1999). Cette technique permet d'avoir un diagramme de GANTT représentant les états, et ce, en gardant une bonne fluidité, peu importe la taille de la trace.

La technique qu'utilise Jumpshot a été introduite par Chan et al. (2008) afin d'afficher graphiquement une certaine portion de la trace. Pour ce faire, l'auteur utilise le concept de rectangle englobant pour séparer la trace. Les rectangles sont insérés dans un arbre binaire de façon à ce que la racine englobe toute la trace et ses enfants, puis les enfants de la racine chacun une moitié et ainsi de suite. Ces informations sont contenues dans un fichier de trace de format nommé SLOG2. Lorsqu'on veut afficher une partie de l'information, on prend toutes les boîtes englobantes qui se chevauchent, donc le temps d'accès aux données est réduit, puisque la recherche dans l'arbre est rapide.

La génération de ce fichier se fait, tout comme dans le cas de la structure dédiée présentée ci-haut, en ordre chronologique. Par contre, le fait que ce type de fichier soit conçu pour l'affichage, il est donc étroitement lié au visualiseur, ce qui en fait une solution moins flexible. De plus, la trace doit contenir l'information de l'état du système. Ce qui fait en sorte que le traceur aura un impact plus grand sur le programme tracé.

2.2.4 Analyse de trace en ligne

Jusqu'à présent, les techniques d'analyse proposées étaient faites *a posteriori*. Il existe aussi des analyses que l'on peut faire en ligne. Ces analyses sont utiles pour récolter des statistiques sur notre système. L'analyse d'un flux de données à l'aide d'une modélisation cubique (Ezzati-Jivan and Dagenais, 2014) permet de calculer différentes statistiques, et par la suite faire des requêtes, sur n'importe quel intervalle de temps pour obtenir l'information voulue. Cette technique permet de collecter des statistiques sur différentes dimensions d'un système (ex. la mémoire, les processus ou les fichiers). On peut calculer des sommes, des moyennes et faire le décompte. Par exemple, on peut calculer le nombre d'occurrences d'un événement spécifique, le débit d'écriture et de lecture d'un fichier ou encore la moyenne d'utilisation d'un CPU.

Les composantes principales de cette solution sont l'arbre de dimension et le cube qui est sauvegardé dans une base de données. Les arbres représentent chacun une dimension du

système. Les valeurs de l'arbre serviront entre autres comme clé pour les requêtes dans la base de données. Les cubes représentent un intervalle de temps et ils contiennent les arbres. À chaque intervalle de temps défini, on calcule les statistiques et on insère les valeurs dans un cube. Il est possible de choisir la granularité de ces intervalles de temps. Par exemple, pour une trace de cinq minutes, les intervalles peuvent être d'une seconde tandis que, pour une trace d'une heure, on peut avoir des intervalles de cinq minutes.

2.2.5 Visualisation de trace

À la suite des analyses de traces, il est important d'avoir les vues appropriées afin de bien comprendre l'information récoltée. Il est possible d'afficher les données brutes de la trace. Par contre, lorsque le programme tracé devient complexe, il est alors plus difficile d'avoir une vue d'ensemble du déroulement de l'exécution de son application afin de déceler un problème de performance.

Il existe plusieurs façons de montrer les résultats d'une analyse. Par exemple, on peut utiliser un histogramme pour montrer la densité d'événements ou des graphiques basés sur l'utilisation du CPU. On peut aussi utiliser le diagramme de Gantt, qui est employé dans la plupart des logiciels de visualisation de traces. Ce type de diagramme est parfait pour montrer les tâches d'un système en fonction du temps. Par exemple, Trace Compass utilise ce type de graphique, avec le gestionnaire d'état, pour montrer l'état de chaque fil d'exécution en fonction du temps. La figure 2.2 montre ce diagramme ainsi que l'histogramme des événements dans le logiciel Trace Compass. Jumpshot et les outils de visualisation développés par Microsoft (Windows Performance Toolkit (WPT)) utilisent aussi les diagrammes de Gantt de la même façon.

Figure 2.2 Vues du logiciel Trace Compass

Dans la littérature, de nombreuses façons ont été montrées pour optimiser l'affichage de cette vue appliquée au traçage. Jumpshot utilise son propre format de trace (Chan et al., 2008) fortement lié à cette vue. Il y a aussi Ezzati-Jivan and Dagenais (2012) qui ont présenté une façon de réduire le nombre d'états affichés en utilisant des états synthétiques, qui en regroupent plusieurs, de façon à ce que, lorsque l'on zoom, le niveau de détail change.

2.3 Langages de modélisation

Dans la section précédente, nous avons abordé plusieurs techniques d'analyse et de visualisation de traces. Par contre, une limitation importante de ces solutions est le manque de flexibilité. En effet, les outils sont souvent conçus pour utiliser un type de trace et un traceur en particulier. Ceci force le développeur à utiliser des analyses prédéfinies qui ne correspondent pas nécessairement à ses besoins. La solution est de rendre les analyses génériques pour que l'utilisateur puisse définir lui-même les analyses appropriées. Les langages de définition sont un bon compromis entre la flexibilité et la facilité d'utilisation. Ces langages sont largement utilisés dans le domaine de la sécurité et de l'analyse de systèmes.

Dans cette section, il sera question de langages déclaratifs comme Snort et impératifs comme SystemTap ou DTrace. Les langages basés sur les automates ainsi que les langages spécifiques

au domaine (DSL) seront aussi abordés.

2.3.1 Langages déclaratifs

Les langages déclaratifs permettent de décrire ce que l'on veut faire sans toutefois dire la façon de le faire.

Snort

Snort (Roesch et al., 1999) est un des plus importants systèmes de détection d'intrusion (Intrusion Detection System (IDS)) libres qui permet de capturer et d'analyser, en temps réel, le trafic réseau. Il permet de détecter les intrusions en utilisant une syntaxe qui permet de définir des règles de détection. L'utilisateur peut donc définir des règles qui sont propres au contexte dans lequel son système s'exécute.

Les règles peuvent utiliser les différents champs contenus dans les paquets réseau. Elles sont composées de deux parties importantes. La première est la définition de l'action à prendre (journalisation, alerte ou ignorer le paquet) dans le cas où la règle est satisfaite. Cette partie contient le protocole et les adresses IP source et destination. La deuxième partie contient les options qui permettent de définir les champs d'application, comme la direction du trafic et le port d'intérêt.

2.3.2 Langages impératifs

Contrairement aux langages déclaratifs, les langages impératifs permettent d'exprimer la façon d'atteindre l'objectif voulu.

Langages utilisés par DTrace et SystemTap

Précédemment, nous avons abordé différents traceurs ainsi que leur fonctionnement sans toutefois détailler le langage utilisé par deux d'entre eux. Les traceurs DTrace et SystemTap utilisent tous deux un langage de script dont la syntaxe ressemble à celle du C. Ces langages sont conçus pour le traçage. Ils sont donc utilisés pour ajouter des points de trace ainsi qu'exécuter des actions lorsqu'un de ces points est rencontré. Ils ne peuvent pas être utilisés pour concevoir l'analyse de la trace recueillie.

DTrace utilise le langage D pour définir les points de trace et les actions associées. Le langage comporte deux parties. La première est la définition des conditions, sous forme de prédicats au lieu d'utiliser des structures conditionnelles conventionnelles. La deuxième est la définition de

l'action à prendre lorsque la condition est respectée. Une fois le script complété, il est compilé dans un format intermédiaire, puis envoyé au module noyau de DTrace pour être exécuté et commencer le traçage. Beaucoup de détails sur le langage et d'exemples d'application sont présents dans la littérature (Beauchamp and Weston, 2008; Gregg and Mauro, 2011).

SystemTap possède aussi deux éléments importants dans le langage. Il y a la partie qui défini le point de trace déjà existant que l'on veut activer, ou l'insertion de nouveaux points de trace. Puis, comme avec DTrace, il y a le code de l'action à prendre lorsque le point est rencontré. Le langage de SystemTap utilise une syntaxe plus près de celle du C et utilise les mêmes structures conditionnelles que le langage C. Le script est transformé en C afin de l'utiliser comme module noyau. Par la suite, ce module communique avec SystemTap pour récolter les informations sur le système. Plusieurs exemples d'utilisation sont aussi disponibles dans la littérature (Prasad et al., 2005; Eigler and Hat, 2006).

2.3.3 Langages basés sur les automates

Ce type de langage permet de décrire un automate fini afin de résoudre un problème avec un nombre fini d'états et de transitions.

STATL

STATL (State Transition Analysis Technique Language) est un langage permettant de décrire un scénario d'attaque pour utiliser dans un système de détection d'intrusions (IDS) (Eckmann et al., 2002). Les scénarios contiennent entre autres des états et des transitions. Les scénarios représentent une séquence d'étapes qui permet de déterminer si on est passé d'un système non compromis à un système compromis.

Regardons ce langage plus en détail. Les états et les transitions sont les deux éléments importants dans ce langage. Le scénario doit contenir au minimum deux états (initial et final) et une transition. Un état possède :

- Un identifiant afin de pouvoir le référencer dans les transitions.
- Une assertion facultative. L'assertion est évaluée à vraie si elle n'est pas présente.
- Un bloc de code à exécuter si l'assertion est vraie.

Une transition contient :

- Un nom ainsi que la paire d'états que celle-ci connecte. L'état de départ et de fin peut être le même.
- Le type d'événement qui déclenchera la transition.
- Une assertion facultative dans le même style que pour l'état.

— Un bloc de code à exécuter dans le cas où l'assertion est vraie.

Bref, les assertions de l'état et de la transition doivent être respectées pour que le changement d'état ait lieu. Lors d'un changement d'état, les conditions de la transition sont vérifiées avant ceux de l'état. Puis, si les deux assertions sont vérifiées, on exécute le code de la transition avant celui de l'état.

2.3.4 DSL - Langages spécifiques au domaine

Nous avons exploré plusieurs types de langages et d'exemples de ceux-ci. Par contre, afin de répondre à notre problème, aucun de ces langages ne peut être directement appliqué à notre contexte. Afin de développer notre métamodèle de machine à états et ainsi pouvoir l'utiliser avec un outil modélisation, nous avons besoin d'un langage adapté au traçage et à l'analyse de traces à l'aide du gestionnaire d'état. C'est pour cette raison que le langage spécifique au domaine (DSL) constitue une solution. Ce langage se rapproche beaucoup des langages déclaratifs qu'on peut souvent l'associer à ceux-ci. Les DSL sont axés sur un domaine particulier de problèmes, c'est-à-dire que ce type de langage permet de résoudre des problèmes concernant un domaine d'application spécifique. L'article de Van Deursen et al. (2000) fait l'état de l'art des langages spécifiques au domaine. Dans l'article, il donne la description suivante pour les DSL : c'est un langage de programmation ou de spécification qui offre une puissance expressive sur un domaine particulier de problèmes. Il est souvent petit et offre des notions et des abstractions liées au domaine.

Les avantages d'utiliser un DSL sont : il offre un niveau d'abstraction propre au domaine, les experts du domaine peuvent comprendre, valider et développer le DSL et il incarne le savoir du domaine. Les inconvénients liés à son utilisation sont : un coût plus élevé pour concevoir et maintenir le langage et il demande plus d'apprentissage de la part des utilisateurs du DSL comparativement à un langage générique populaire comme le Java.

Dans cet article, les auteurs ont aussi identifié les étapes pour construire un DSL. Voici les étapes initiales pour construire notre langage.

1. Identifier le problème.
2. Rassembler le savoir du domaine d'application.
3. Grouper le savoir en notions et en opérations.
4. Concevoir le DSL.

2.4 Modélisation

Les langages spécifiques au domaine présentés ci-dessus sont de bonnes bases pour répondre à notre problème. En effet, l'utilisation d'un langage personnalisé permet de concevoir des analyses pertinentes aux problèmes de performance que nous tentons de résoudre avec le traçage. Par contre, l'écriture et l'apprentissage d'un nouveau langage peuvent s'avérer décourageants pour un nouvel utilisateur. Il aura tendance à utiliser les outils et les analyses déjà présents dans le traceur ou le logiciel d'analyse.

La modélisation est une façon de représenter graphiquement ce que l'on veut faire. Par exemple, la modélisation du comportement d'un système. En combinant la modélisation et les DSL on peut obtenir un outil adapté au besoin du domaine.

2.4.1 Outils de modélisation

Dans cette section, nous aborderons les outils de modélisation présents sur le marché. Certains d'entre eux utilisent les standards de modélisation comme le UML, et d'autres sont conçus pour être utilisés avec un DSL. Par exemple, Graphiti et Sirius sont des cadriciels permettant de bâtir son propre outil de modélisation. D'autres logiciels, comme Papyrus et la suite *Rational Rose*, sont des outils utilisant UML.

Cadritiels de modélisation

Les cadriciels présentés dans cette section utilisent la plateforme Eclipse, car elle est libre. Les deux outils mentionnés plus haut, soit Graphiti et Sirius, évoluent autour d'Eclipse Modeling Framework (EMF) [2] (Steinberg et al., 2008) qui est le cadriciel qui permet de construire des outils de modélisation basés sur un modèle de données standard utilisé par plusieurs applications.

Graphical Modeling Framework fut le premier cadriciel graphique à utiliser les composantes de EMF. Le développement n'est plus très actif en raison des alternatives plus simples et plus complètes.

Graphiti [3] est un cadriciel graphique, relativement récent, qui permet de développer un éditeur de diagramme et supporte les modèles créés avec EMF. Il se veut être une alternative à Graphical Modeling Framework (GMF) grâce à sa simplicité d'apprentissage. Dans ce sens, un des objectifs de Graphiti est de fournir une interface de programmation (API) simple

2. `https://www.eclipse.org/modeling/emf/`
3. `http://eclipse.org/graphiti/`

et facile d'utilisation afin de cacher l'implémentation des technologies spécifiques à Eclipse comme GEF (Graphical Editing Framework) et Draw2D. L'architecture de Graphiti (Brand et al., 2011) permet de le rendre flexible à plusieurs niveaux. De plus, l'article de Stritzke and Lehrig (2013) montre que Graphiti a un grand potentiel pour remplacer GMF.

Dans la même catégorie que Graphiti, Sirius [4] (Viyovic et al., 2014) se veut aussi un cadriciel graphique pour simplifier la tâche de création d'éditeurs de diagrammes. L'inconvénient avec Graphiti est l'obligation de connaître la programmation orientée objet en Java et les extensions Eclipse afin de pouvoir l'utiliser. Sirius quant à lui, est développé par-dessus GMF et ne constitue pas un remplacement total de celui-ci. Cette conception permet d'éviter la programmation en utilisant des outils graphiques pour définir le modèle à utiliser dans l'éditeur que l'on veut créer.

Ces deux outils de conception permettent d'arriver au même résultat en utilisant différentes technologies et différentes façons de penser, mais en gardant EMF comme base (Vujović et al.).

Papyrus

Papyrus [5] (Gérard et al., 2010) est un environnement de modélisation, basé sur la plateforme Eclipse, pour éditer des modèles EMF, mais plus particulièrement le UML. Il respecte la norme UML telle que définit par l'Object Management Group (OMG) et supporte aussi différent langage de modélisation comme SysML ou MARTE. Papyrus permet même de créer des DSL par l'entremise du mécanisme de profils introduit dans UML.

Papyrus est un outil à part entière. Il ne permet pas de créer son propre outil de modélisation comme Graphiti ou Sirius. Il est conçu pour éditer différents modèles de manière graphique, textuelle ou en arbre. Papyrus utilise des technologies similaires aux autres outils présentés et il est libre. L'ouverture de celui-ci permet entre autres d'offrir Papyrus pour des domaines plus spécifiques comme le temps réel avec Papyrus-RT [6].

Rational Rose

Contrairement aux autres solutions proposées précédemment, la suite de logiciel *Rational Rose* [7] n'est pas à code source ouvert. Cette famille de produits créée par IBM permet de faire de la modélisation UML. Ces produits sont très complets et touchent plusieurs domaines

4. `https://eclipse.org/sirius/`
5. `https://www.eclipse.org/papyrus/`
6. `https://projects.eclipse.org/projects/modeling.papyrus-rt`
7. `http://www-03.ibm.com/software/products/en/ratirosefami`

de l'informatique. La suite d'outils comprend entre autres des outils pour la conception de logiciels, de bases de données et de systèmes temps réel. Par contre, le développement s'est arrêté, mais il est encore utilisé par certaines compagnies. C'est pourquoi Papyrus est apparu afin de remplacer cette solution désuète.

2.4.2 UML

Les outils de modélisation et les travaux en modélisation utilisent ou font souvent référence au UML. Celui-ci est un langage de modélisation unifié qui fournit un standard afin de visualiser la conception d'un système. Cette spécification a été proposée et est maintenue par l'OMG. La spécification UML (Object Management Group (OMG), 2013) est complète et largement utilisée depuis plusieurs années.

Il sera question plus précisément du diagramme de machine à état dans cette section.

Machine à état

Le diagramme de machine à état fait partie des modèles comportementaux de UML. Ces modèles décrivent le comportement et les interactions d'un système. Le concept de comportement permet de modéliser des changements dynamiques et les comportements sont définis par des événements qui se produisent dans le temps. La définition fournie par la spécification UML pour la machine à état est la suivante : une ou plusieurs régions qui contiennent un ensemble de sommets interconnectés par des arcs représentant des transitions. La figure 2.3, tirée de la spécification UML, représente cette machine à état.

Dans la description, on mentionne le terme *région*. Une région est un fragment de comportement de la machine à état. Plusieurs régions peuvent s'exécuter parallèlement à condition qu'elles soient dans la même machine à état. De plus, chaque région doit posséder un état initial et final. Les régions contiennent des sommets et des transitions. Les sommets sont une abstraction de différents types de noeuds. Les noeuds sont de type *état* ou *pseudo état*. Un sommet peut être la source et la destination de plusieurs transitions.

Les états d'une machine à état peuvent se retrouver sous trois formes. La première est la forme simple. La deuxième est un état composite. C'est-à-dire que l'état peut contenir une région. La dernière forme est la possibilité d'avoir une machine à état complète à l'intérieur d'un état. Dans notre cas, nous accorderons plus d'importance à la forme simple. Ces états peuvent avoir des comportements associés. Les comportements peuvent être définis comme entrée, sortie ou interne (*doActivity* dans la spécification). Dans les deux premiers cas, le comportement sera exécuté, respectivement, à l'entrée et à la sortie de l'état. Le comportement interne sera

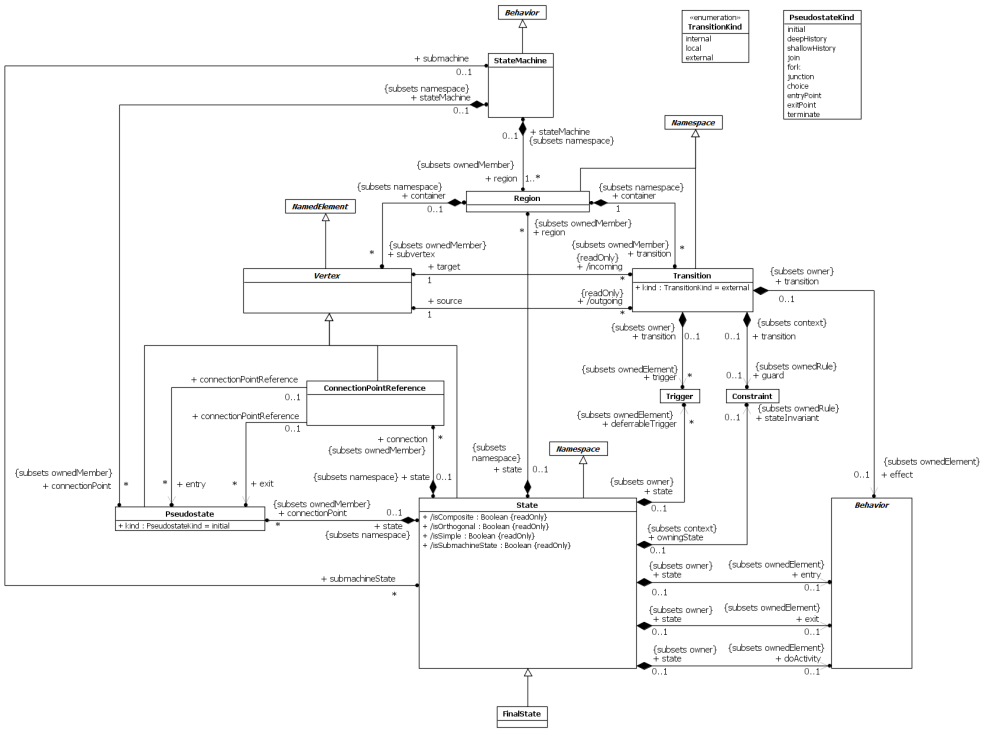

Figure 2.3 Diagramme illustrant la structure logique de la machine à état en UML

exécuté après le comportement d'entrée. Un cas spécial d'état est l'état final. Celui-ci indique la complétion d'un comportement et, par le fait même, indique que la région est complétée. Les états et l'état final sont considérés comme des sommets stables, car il reste dans l'état tant et aussi longtemps qu'il n'y pas d'événement qui déclenche le changement.

Les pseudo-états sont des sommets dits transitoires. On ne s'arrête pas, on continue la transition. L'état initial d'une machine à état fait partie de cette catégorie. Il est unique dans sa région et est la source d'au moins une transition. À l'exception de l'état initial, ces états sont généralement utilisés pour composer ou séparer des transitions. Par exemple, les pseudo états `Join` et `Fork` permettent respectivement de faire un travail de synchronisation de plusieurs transitions et de séparer une transition en plusieurs autres. Un autre pseudo-état qui aura de l'importance plus tard est `Choice`. Ce pseudo-état est un branchement conditionnel. Il peut séparer la transition en différentes transitions dépendamment de la décision, qui est évaluée dynamiquement.

En ce qui concerne les transitions, ce sont des flèches dirigées provenant d'une seule source vers une cible (sommet). La source et la cible peuvent être le même sommet. La transition doit contenir un événement déclencheur et peut aussi avoir un comportement associé à celle-ci.

Il existe d'autres concepts de la machine à état spécifiés par le UML qui ne sont pas abordés

dans cette section pour des raisons de non-pertinence pour la solution proposée. Avec tous les concepts définis par UML, il est possible de modéliser la quasi-totalité des comportements d'un système.

Comme nous pouvons le constater, plusieurs solutions ont été proposées pour résoudre différents problèmes concernant plusieurs aspects du traçage. Par contre, peu d'entre elles offrent la flexibilité à l'utilisateur de personnaliser facilement ses outils de travail, afin de permettre de résoudre des problèmes spécifiques à sa situation et à son environnement. De plus, la modélisation, avec la spécifiaction UML, est présente depuis plusieurs années et est exploitée dans différentes sphères du logiciel. Cependant, la combinaison du traçage et de la modélisation n'est pas présente dans la littérature. Pour y remédier, nous avons conçu un outil permettant de combiner ces deux technologies. La problématique ainsi que les démarches entreprises afin d'atteindre nos objectifs seront présentées au chapitre suivant.

CHAPITRE 3 MÉTHODOLOGIE

Cette section traitera des démarches entreprises afin de répondre à la question initiale et afin de remplir les objectifs de recherche fixés précédemment. Tout d'abord, un rapide rappel de la problématique sera fait, puis nous enchaînerons sur la démarche utilisée.

3.1 Rappel de la problématique

Les traceurs modernes offrent une grande flexibilité à l'utilisateur afin de les aider à trouver des problèmes de performance spécifiques à leurs systèmes ou applications. Par contre, les analyses présentes dans les différents logiciels d'analyse de traces sont peu flexibles, du moins difficiles à personnaliser ou à créer. Les développeurs ont besoin de plus de flexibilité lors de la conception d'analyses.

Trace Compass utilise présentement un système d'analyse dirigé par les données pour remédier à ce problème. Cette technique utilise un fichier XML pour définir les différentes facettes d'une analyse. Elle permet de créer des analyses adaptées au besoin de l'utilisateur. En contrepartie, l'utilisation de ce nouveau langage ajoute de la complexité au processus de création d'une analyse. Nous avons donc besoin d'un outil facile à utiliser afin de capturer toutes les informations d'une analyse et ainsi rendre l'accès à la personnalisation plus facile pour les développeurs.

3.2 Démarche

Dans le but d'atteindre les objectifs, nous avons établi certaines étapes clés afin d'obtenir un produit fonctionnel et prouver la faisabilité ou non de mon interrogation.

Comme notre recherche est la continuité de travaux antérieurs, l'analyse XML en particulier, il faut effectuer un transfert de connaissances. En effet, il est essentiel de récolter toute l'information nécessaire sur les analyses en général et sur l'analyse faite par XML, pour nous permettre d'avoir une vision globale des techniques utilisées. Cette étape consiste à utiliser et à comprendre le travail qui a été fait dans l'analyse XML. Pour ce faire, nous avons discuté avec les personnes qui ont travaillé sur cette technologie et avec celles qui utilisent et comprennent l'analyse XML (dans notre cas, l'équipe de Trace Compass)

La prochaine étape de mon cheminement consiste à en apprendre davantage sur les machines à état, puisque notre travail porte sur l'utilisation de celles-ci. Puisque nous devons concevoir

un nouveau modèle adapté au domaine du traçage, nous avons étudié les différents standards existants, typiquement UML. Cette étape a permis d'avoir une base solide lorsqu'est venu le temps de concevoir le modèle.

Étant donné que les tests s'effectueront dans Trace Compass, c'est-à-dire sur la plateforme Eclipse, la maîtrise et l'apprentissage de certaines technologies sont importants. Des recherches sur les différentes options disponibles dans le domaine de la modélisation ont été effectuées. Nous avons étudié des technologies comme EMF, GMF, Graphiti et Sirius.

À la suite de ces recherches, nous avons fait une phase de prototypage. Étant donné qu'il y a deux parties à nos travaux, nous avons fait deux prototypes distincts afin d'évaluer les deux solutions sans qu'ils interfèrent l'un sur l'autre. La première partie est le modèle de la machine à état et la deuxième partie est l'utilisation des cadriciels graphiques. En ce qui concerne la technologie utilisée pour manipuler et concevoir notre modèle, nous avons choisi d'utiliser EMF, car il est la base de la modélisation dans Eclipse. Pour les cadriciels, GMF, Graphiti et Sirius ont été testés.

Le cadriciel graphique a été choisi en fonction des critères suivants :

- Facilité d'utilisation et de développement.
- Communauté de développement active.
- Possibilité d'intégration dans Trace Compass.
- Interface graphique compréhensible par l'utilisateur.

À l'aide de ces critères, Graphiti s'est avéré le meilleur choix dans notre situation. En effet, il est facile à utiliser lors du développement, la communauté répond rapidement aux questions posées, il s'intègre facilement à Trace Compass et l'interface est moderne et facilement personnalisable. Une fois le cadriciel choisi, nous avons fait une version améliorée du modèle qui sera utilisé avec Graphiti pour faire un outil graphique complet en utilisant ce modèle.

Afin de valider si la solution est correcte et convient au besoin de l'analyse, nous devons avoir une base qui servira de point de référence pour la validité du modèle. Comme notre solution doit au moins avoir toutes les fonctionnalités de l'analyse XML, nous avons utilisé des analyses précédemment conçues par cet outil et qui ont été validées par l'équipe de Trace Compass. En sachant que ces analyses sont valides, elles ont été reproduites dans notre outil graphique et nous avons vérifié que le résultat est le même. Par contre, il est necessaire de bien choisir les analyses de références. Il est important que ces analyses couvrent toutes les fonctionnalités de base et les cas plus complexes qui peuvent exister dans l'outil précédent.

Les différentes solutions explorées et les résultats de nos travaux sont expliqués plus en détail dans l'article de la section suivante.

CHAPITRE 4 ARTICLE 1 : INTEGRATED MODELING TOOL FOR INDEXING AND ANALYZING STATE MACHINE TRACE

Authors
Simon Delisle
École Polytechnique Montréal
simon.delisle@polymtl.com

Naser Ezzati-jivan
École Polytechnique Montréal
n.ezzati@polymtl.ca

Michel Dagenais
École Polytechnique Montréal
michel.dagenais@polymtl.com

Submitted to: Journal of Computer and System Sciences

Keywords: Performance analysis, Tracing, State machine, UML, Modeling, Model-driven.

4.1 Abstract

It is important to understand an application or system behavior in order to identify potential performance problems. Tracing is a good starting point to get this information. A trace provides a lot of information about an application run-time behavior. However, some preprocessing and analysis is often required to extract or highlight valuable information. Since the analysis plays an important role in the process, it should be supported adequately. Most of the time, a number of built in analyses are available in trace analysis and viewing tools. They are often adequate and give a good picture of the application execution. However, the need for more flexible analyses often arises. In this paper we present a solution to overcome this problem. A modeling tool is proposed to more easily define or modify the trace analyses. It can be used to define how to model the state of the traced system, and how to define filters and patterns. In this way, many of the advanced trace analysis tasks become much simpler to specify. In this paper, real use cases, from different levels and categories, are modeled using our modeling tool to show its genericity and expressiveness to model an execution trace.

4.2 Introduction

Using conventional techniques, such as static analysis or debugging, to find performance issues in distributed and multi-core systems is difficult and almost impossible to do in interactive mode. This is why execution traces, with minimal impact on the system studied, provide a good solution. It generates highly detailed data that can be used in sophisticated offline analysis.

Inserting tracepoints in a program or a system is necessary in order to trace a system and diagnose performance problems. Those tracepoints will generate trace events, whenever encountered as the program runs. Linux Trace Toolkit next generation (Desnoyers and Dagenais, 2006), DTrace (Cantrill et al., 2004), SystemTap (Prasad et al., 2005) and Perf (Edge, 2009) are some of the modern Linux operating system tracers. Traces can be generated for both kernel and user space software.

However, the raw trace itself is not necessarily of much help, since the amount of collected data can be huge. This is why we need an efficient tool to analyse the trace and provide us with the information we need. Many different trace analysis software tools are available, including Trace Compass [1], Jumpshot [2] and Microsoft performance analysis tools [3]. One important limitation of many of these tools is the flexibility. They often constrain the user to a specific type of trace, and to use only a few builtin analyses and views.

Nonetheless, even if these analyses cover a lot of cases, we often need more information to solve our particular problem. For this reason, Trace Compass introduced a flexible analysis using an XML syntax (Wininger, 2014) (Kouame et al., 2015). It is thus possible to describe how the events modify the modeled state of the traced system, and to specify filters and patterns based on the events and the modeled state. The remaining problem with this solution is that you still need to write large XML files, and to learn a new syntax to achieve your goal. This is why we propose, in this paper, a modeling tool to capture in a convivial way all the information related to trace analysis. The main contribution of this paper is to combine a state-of-the-art kernel and user-space tracing system with powerful, yet simple to use, modeling. In addition, we demonstrate how it was used to easily specify detailed kernel analysis for real programs running on Linux systems, as well as to uncover real problems in complex applications, including the Eclipse Trace Compass tool. In this paper, we base our work on Trace Compass and LTTng, to generate kernel traces to model the main behavior of the Linux operating system.

1. `http://projects.eclipse.org/projects/tools.tracecompass`
2. `http://www.mcs.anl.gov/research/projects/perfvis/software/viewers/`
3. `https://msdn.microsoft.com/en-us/library/hh448170.aspx`

The article begins with the related work section and continues with a description of the current and the new proposed architectures. Thereafter, we present the model that will be used in the modeling tool and explain how it works. Afterwards, we discuss interesting applications and use cases of the proposed solution, and finally an outlook of future work.

4.3 Related Work

When it comes to identifying performance issues, we need a complete set of tools that work well together, and help to quickly find our problems. Multiple tools can be combined into one tool for better consistency. These tools provide solutions for tracing our systems, analyze our traces and show us useful views to solve those problems.

4.3.1 Tracing

Tracing involves gathering information on the execution of an application or system while it is running. The data contained in the trace is intended to identify performance issues. For the trace to be as close to reality as possible, the tracer needs to have almost zero impact on the traced system, not disturbing the normal execution of our application. Unlike debugging, tracing does not interrupt the execution.

In order to trace your program, you need to insert tracepoints at important locations. This can be done statically or dynamically. In the first case, we need to add tracepoints manually using the TRACE_EVENT (Rostedt, 2010) macro for Linux. For this, we need to have access to the source code and have the possibility to recompile the traced program. In the second case, we can add tracepoints without recompiling, tracepoints are added dynamically during the execution of our program. Kprobe (Goswami, 2005) allows doing this for the Linux kernel.

We have two types of traces: kernel and user-space traces. In the next section, I will focus on Linux tracing. For Windows systems, ETW (Insung Park, 2007) is the proprietary tracing platform proposed by Microsoft. It can trace both kernel and user-space. One particularity of this tracer is that you can save the initial state of the system before you start tracing. This provides more information when you want to analyze the trace later.

Kernel tracing

SystemTap was developed by Red Hat in 2005 (Prasad et al., 2005). It was designed to facilitate kernel tracing for system administrators. SystemTap is a scripting language that

can use the TRACE_EVENT macro for static instrumentation, and kprobes for dynamic instrumentation. This tracer is a good choice to diagnose performance problems. It can provide statistics during the program execution and, due to its scripting language, this tracer is flexible. However, SystemTap traces files can become very large. SystemTap does not use a compact trace format, everything is written in a text file. Moreover, the analysis is typically done during tracing; it is more difficult to perform additional analyses afterwards. For this reason, SystemTap is not adapted to our situation.

Since 2009 (Edge, 2009), Perf is part of the tools bundled with the mainline Linux kernel. Perf use hardware and software performance counters to provide information on the Linux kernel. It also supports the TRACE_EVENT macro and kprobe for instrumentation. Perf was initially developed for sampling based on performance counters. When a certain limit is reached, an interruption is triggered and the counters values are saved along with context information such as the instruction pointer. With sampling, Perf has a low impact on the traced system, but the collected data is less accurate.

DTrace (Cantrill et al., 2004) was first developed for Solaris. DTrace has zero impact on the system when tracepoints are disabled. To achieve this, it uses dynamic instrumentation. DTrace use its own C-like language, the D language (Gregg and Mauro, 2011), to define actions to execute when a tracepoint is reached and an associated prerequisite condition is true. This language is the same for kernel and user-tracing tracing.

LTTng was introduced in 2006 (Desnoyers and Dagenais, 2006) to trace the Linux kernel with the objective to minimise the impact of tracing on the traced system. LTTng version 2 (Mathieu Desnoyers, 2012) was released as a kernel module and uses the Common Trace Format, an open trace file format (Desnoyers, 2011). It uses tracepoints that are already in the kernel and supports kprobe for dynamic instrumentation. To achieve the best performance, per-cpu lockless circular buffers are used to collect events. The filled buffers can thereafter be written to disk or sent through the network. One advantage of LTTng is that you can add context information to each event. Performance counters are one of the possible elements of context information. Other possible context fields include the current CPU or the current thread Id. This way, you can know which process was running on a CPU when an event occurs.

User-space tracing

Even if a kernel trace can be very useful to detect problems during execution, sometime we need more specific information on the application itself. In the same vein as kernel tracing, user-space tracing needs to minimize the impact on the application. This is an alternative

to traditional debugging with breakpoints.

DTrace and SystemTap also have a user-space component. DTrace uses its own language to define tracepoints, and SystemTap uses uprobes (Jim Keniston, 2010) which is similar to kprobes. These tracers have a significant impact on the traced application. Indeed, they do system calls each time a tracepoint is reached.

LTTng-UST (Fournier et al., 2009) is the user-space component of LTTng, mentioned earlier. The advantage over the other tracers is that it runs entirely in user-space. LTTng-UST uses static tracepoints in the application, like kernel tracing. To insure that the tracer has no impact, events are directly written to shared memory. A consumer daemon is in charge of writing the events to disk. Since LTTng-UST uses shared memory, it remains decoupled from the consumer daemon. If events are produced faster than the consumer can send the filled buffers to disk or to the network, events are discarded (and lost events accounted for) rather than blocking the traced application.

4.3.2 Trace analysis

Since a major goal for tracing is to detect performance issues, we need to build well suited analyses. Analyses are often performed a posteriori, after the trace is recorded. This way, we can perform more costly and precise analyses, without interfering with the execution of our system. There are several possible types of analysis. In this paper, I will focus on state based analysis.

Special types of traces contain all the state information in the trace itself, like SLOG2. For this kind of trace, a large part of the analysis is fixed and was done at trace generation time, determining for each state the start and end time. It is thus easier to quickly show the state in a trace viewer. Jumpshot (Zaki et al., 1999) allows showing SLOG2 traces in Gantt style views. The advantage to have states in traces, with a suitable random access format, is that views may be quickly displayed, no matter the size of the trace. The drawback of this technique is the fact that the definition and granularity of the states is fixed. The trace is made for a specific visualization purpose and is bound to the viewer.

Other techniques like synthetic events generation (Ezzati-Jivan and Dagenais, 2012) or multilevel analysis (Ezzati-Jivan and Dagenais, 2014) can be used for online analysis and provide us with good metrics on the execution of our system. In our case, another solution, related to the method used in Trace Compass, is more appropriate for this work.

State system

The principle behind this method is to model the state of the traced system from the events in the trace. The goal is to transform events into state changes. It uses a *state provider* to achieve this. The state provider indicates how to convert an event into a state. The modeled state is built in chronological order, the first time you open the trace. With this modeled state and a dedicated data structure to store it, we can make queries anywhere in the trace to quickly obtain the state at this moment. This unique data structure to store the state history, presented by (Montplaisir-Gonçalves et al., 2013), (Montplaisir et al., 2013) and (Ezzati-Jivan and Dagenais, 2013), is a tree structure to insure a logarithmic search time. It stores state values as intervals. The particularity of the tree is that it does not need to be rebalanced since the construction is made in chronological order, and does not allow insertions in the past.

4.3.3 Languages

Since we want to bring more flexibility to trace analysis, we need to simplify the definition of these analyses by the user. Descriptive languages may be a good solution for this problem. These languages are widely used in computer.

Different types of languages exist for different purposes. Declarative languages describe what you want to do and not how to do it. Snort (Roesch et al., 1999) is a good example of such languages. It is one of the most important open source network intrusion detection system. It is based on a collection of rules defined by the user through a declarative language.

Imperative languages describe commands to execute. As mentioned previously, DTrace (Beauchamp and Weston, 2008; Gregg and Mauro, 2011) and SystemTap (Prasad et al., 2005; Eigler and Hat, 2006) use their own language to define how to react to tracepoints. These two languages are an example of imperative languages.

We can also have automata based languages like STATL (State Transition Analysis Technique Language) (Eckmann et al., 2002), a language to describe attack scenarios for an intrusion detection system.

The problem with these types of languages is that they are not adapted to a specific subject like tracing. We need a language that fits the requirements of trace analysis, based on the state system. A solution that is widely used in the modeling world is Domain Specific Languages, DSL. As the name suggests, a DSL is used to solve specific problems in a domain. Van Deursen et al. (2000) gives a good definition and overview of DSL. The advantage of this solution is that it can be expressed at the level of abstraction of the problem. It can also

be understood, validated and even developed by domain experts. On the other hand, DSL are more costly to design and maintain, and DSL users need to put more effort to learn a new language, compared to a general purpose language like Java.

4.3.4 Modeling

To mitigate the problem of learning a new language, it is possible to use modeling to achieve a more user-friendly environment. By combining DSL and modeling, we can obtain a well adapted yet easy to understand tool. Many technologies can be used to build such a tool. Since our experimentation will be performed with Trace Compass, based on Eclipse, we looked at Eclipse based modeling tools. However, there are other tools such as Rational Rose [4]. Within Eclipse are available modeling frameworks like Graphiti and Sirius, or complete tools like Papyrus, that use UML as model.

In Eclipse, all modeling frameworks are based on the Eclipse Modeling Framework [5] (Steinberg et al., 2008). EMF provides a solid foundation for model manipulation. Graphiti and Sirius use EMF to achieve their goal. Graphiti [6] is a relatively new graphical framework which aims to replace GMF. GMF is an old graphical framework and the community is not very active. Graphiti is interesting, being very flexible due to its architecture (Brand et al., 2011).

Sirius [7] (Viyovic et al., 2014) is a graphical framework that does not require any programming to build a modeling tool. In this paper, we used Graphiti because it is more flexible and easy to integrate in Trace Compass.

In summary, the previous solutions are not really flexible in terms of defining analyses, the type of trace events you can analyse and the problems you can solve. Our solution is more generic providing more flexibility to users when comes the time to define analyses. Furthermore, it is not limited to a specific type of trace and it can be used for any type of trace events. Our tool can be used to solve problems at any level. For exemple, it can be used to analyse kernel execution as well as user-space programs or even a single module within a large system.

4. http://www-03.ibm.com/software/products/en/ratirosefami
5. https://www.eclipse.org/modeling/emf/
6. http://eclipse.org/graphiti/
7. https://eclipse.org/sirius/

4.4 Architecture

As mentioned previously, the current way to analyze a trace in most tools is often hard coded, and sometime it is more difficult to find your particular problem with the provided analysis. Trace Compass has the possibility to define, with an XML file, custom analyses and views. The problem with this solution is that you need to write a complex XML file, and at the same time learn a new syntax and schema. In addition, you currently need to quit the tracing tool environment to add or update these files.

The approach previously used to analyze traces is, as shown in Figure 4.1, to use trace events by parsing and analyzing them to calculate some statistics, to build a state system or to create synthetic events. This information is then used to populate views. In Figure 4.2, we can see that XML specifications were added between the two main components. In the case of Trace Compass, XML is used to specify the state provider that will be used to build the state system, as explained earlier. The proposed solution aims to simplify the process of trace analysis. The modeling tool plays a role to specify how the user wants to convert trace events into states. Later, the user will be able to define synthetic events as well. At the same time you build your analyses, views are automatically generated based on the input model. Figure 4.3 illustrates the architecture of the proposed solution. Basically, the overall workflow is: trace the system, create, by hand, the state machine that represent the model of the state provider and run the resulting analyses.

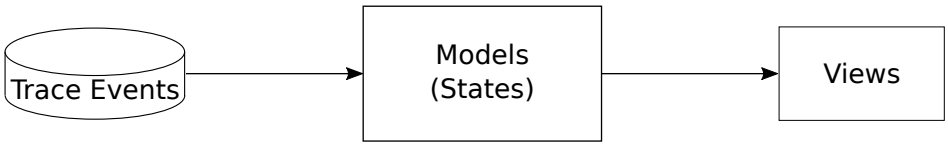

Figure 4.1 Traditional trace analysis approach.

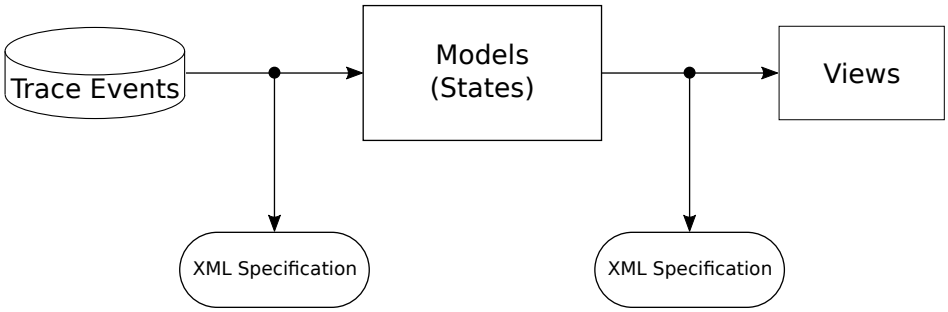

Figure 4.2 Trace analysis with XML specification.

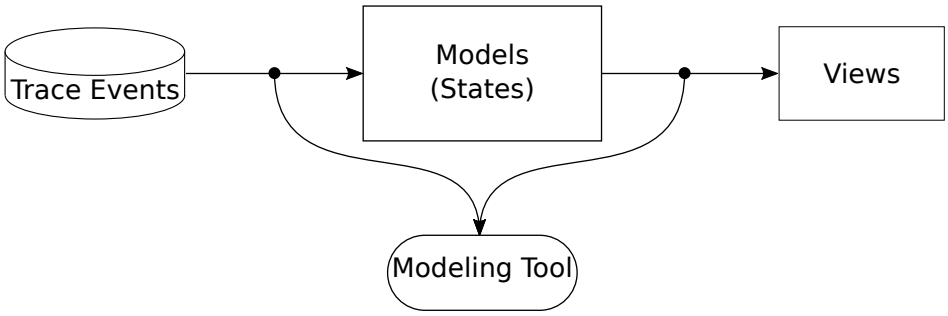

Figure 4.3 Trace analysis with the proposed modeling tool.

4.5 Model Specification

The proposed solution is to use a modeling tool to define trace analysis. A big part of an analysis is to convert events into something that can be used later in the analysis phase. You want to define an action or a state change to trigger when a certain event or pattern happens. The best way to do that is to use a state machine.

UML is a widely used and standard modeling language which already offers support for state machines. We thus based our work on UML, but only used a subset, specialized for execution trace analysis. We could have used full UML profiles to build our modeling tool, but the models would have been needlessly complex for the user. We thus selected a simpler model, more suitable for trace analysis.

4.5.1 Modeling tool

For the modeling part, Graphiti [8] is used for the graphical part, and EMF underneath to save and manipulate the model. The modeling tool is currently a plug-in for Trace Compass. Since the tool is well integrated into Trace Compas, you do not need to use another software to define the analysis, unlike with the other two architectures.

The state system is defined as a model container to store all user defined models. As mention earlier, it uses a disk based and tree oriented data structures, the State History Tree, to efficiently store and query the models. In the modeled state system, the attributes are organized in a tree structure and they are accessible through a unique path, like in a file system. The tree is built as the analysis is running. It contains attributes that describe system resources. Figure 4.4 presents an example of this tree. Since the attribute tree plays a big role in the creation of state changes in the analysis, a tree editor was integrated to build

8. `http://eclipse.org/graphiti/`

the tree beforehand and use it in the diagram. In the model, each state change is defined using the attribute tree. Our modeling tool proposes a way to model the trace events and to store this in the state system, to be used later for analysing the different aspects of the given trace data.

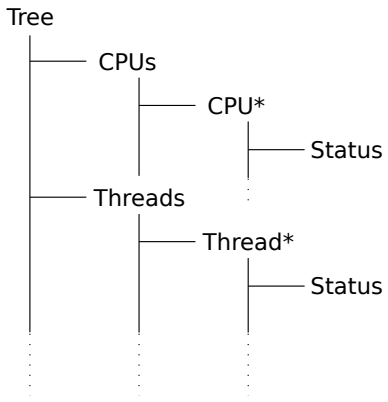

Figure 4.4 Example of an attribute tree

4.5.2 State machine

The state machine is the starting point to build an analysis. It contains the states and transitions that are required for the analysis. You can have multiple state machines in a single diagram. A state machine is not just a container component. In fact, each state machine represents a system attribute that will evolve as the analysis is executed. Figure 4.5 shows two simplified state machines, built with our tool, that will be used for the Linux kernel trace analysis. The first one represents the CPU status and the second one represents the thread status. This model element contains the information about the attribute tree and the state values stored within.

4.5.3 State

A state represents a possible value for an attribute. In Figure 4.5, a CPU can be in user mode or in kernel mode, for example, because of a system call. The state element also specifies the color to use when views will be generated from the diagram. In addition, the state lists the outgoing transitions. This information can be used in the future, for example to do some validation.

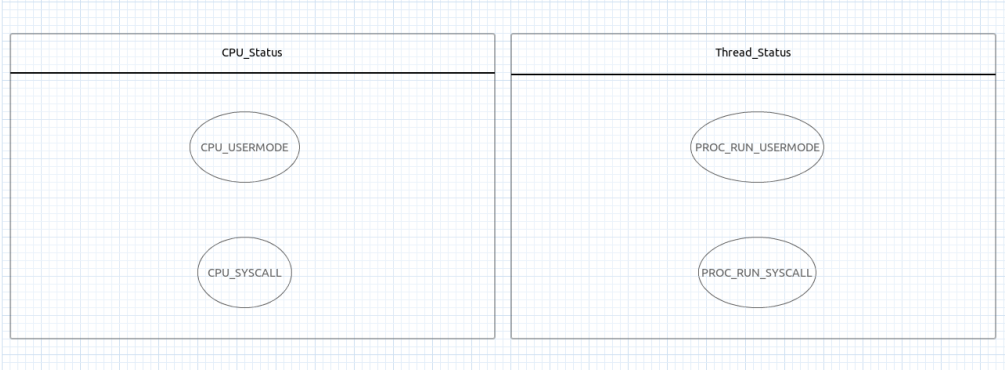

Figure 4.5 State machine and state that will be used for the Linux kernel analysis

4.5.4 Transition

Now we need a way to define the actions that occur when a certain event comes. To do that, we use the transitions. In the diagram, the name of the transition represents the event that will trigger this transition between two states. A transition also contains an element called state change. A state change has two parts: the attribute that you want to change and the new value. It is a key-value relation. Each transition needs to have at least one state change to be valid. There is no limit on the maximum number of state changes. When you create a transition, a state change is automatically added to accelerate the creation of the analysis. Since we know which attributes are changed by the state machine, we can assign to it the value of the target state. Figure 4.6 is an example of a transition and its associated properties. As you can see, when the CPU status is in the user mode, and event `sys_` occurs, it triggers two state changes: the CPU status and the `syscall` value of the current thread on that CPU.

4.5.5 Condition

In order to enhance our model and to support complex analyses, we need to support conditions. This element is a separate component of the model but can only be used to make conditional transitions, as shown in Figure 4.7. Conditions have one inbound transition and one or two outbound transitions. The outbound transitions can go through another condition to build nested conditions. Figure 4.7 shows these nested conditions. You can define a condition whether by using the information in the event itself or information that is already in the model. The classical boolean operators AND, OR and NOT are available to build conditions. In Figure 4.8 and Figure 4.9 are illustrated, respectively, how conditions based on information already in the model are created and how you aggregate those conditions to

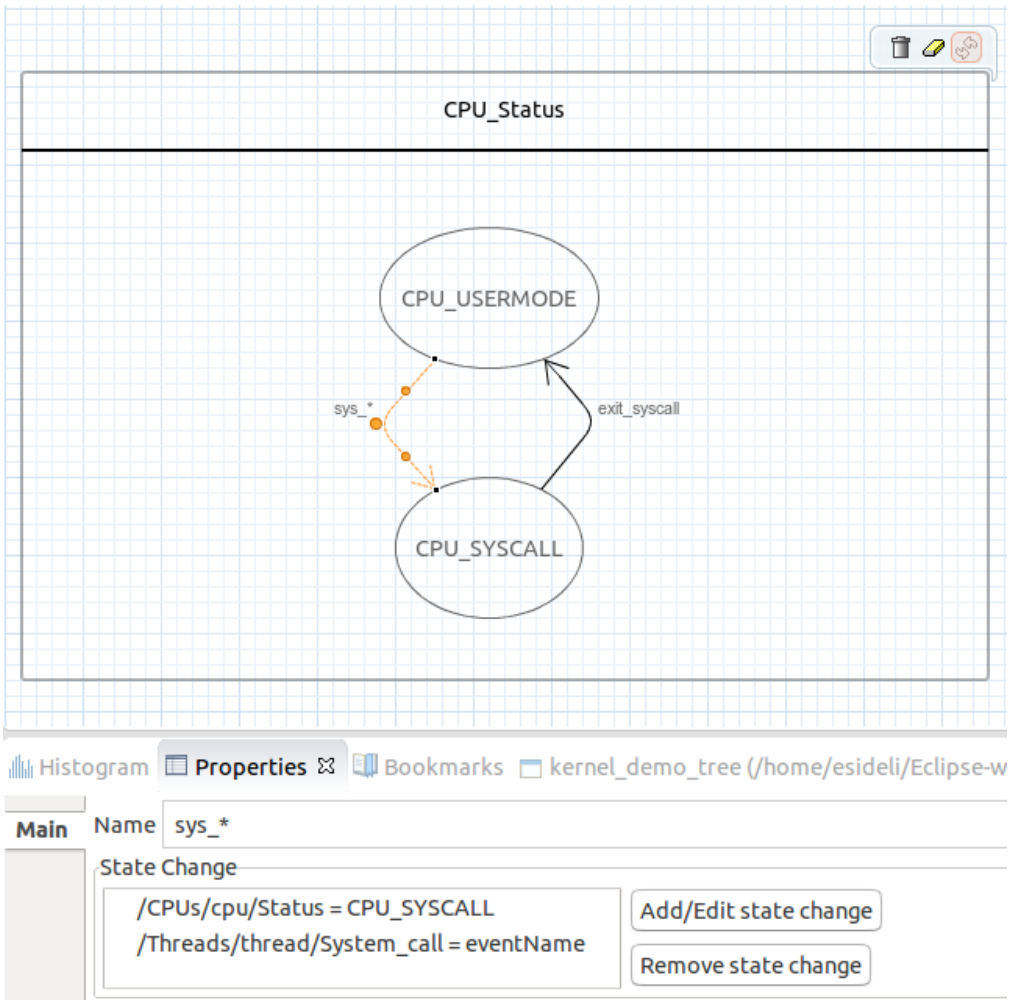

Figure 4.6 Example of transitions that will be used for the Linux kernel analysis

make a more complex condition.

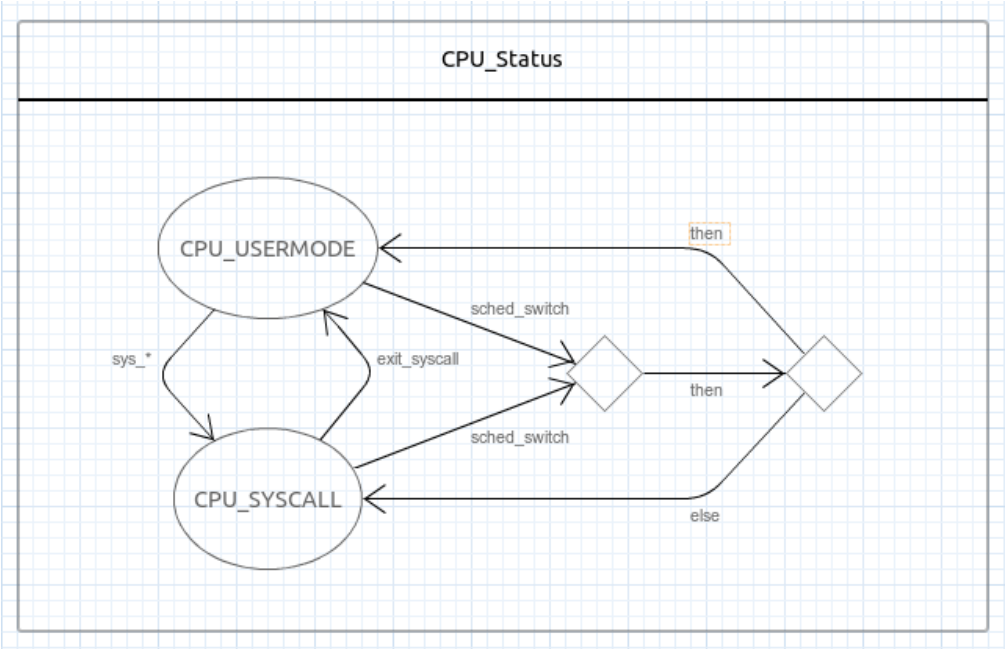

Figure 4.7 Example of condition

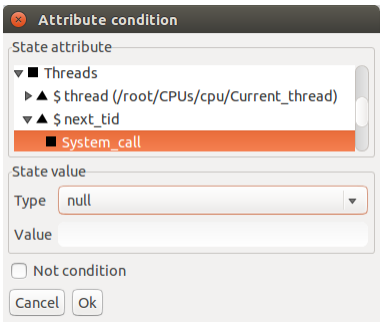

Figure 4.8 Creation of attributes based condition

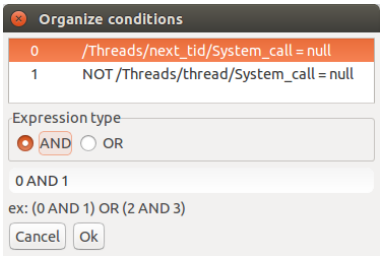

Figure 4.9 Build one condition using single conditions

4.6 Application and Results

In this section, two use cases are presented to show all the possibilities of this tool and how easily an analysis can be built. The first use case is the Linux kernel analysis and the second is a simpler analysis to solve a problem in Trace Compass. We choose these use cases because the Linux kernel analysis is one of the most complex analyses that can be done with the declarative analysis in XML. With this example, we can demonstrate that it is possible to replace the writing of the XML. The second use case shows that you can apply this solution for different kinds of traces.

To simplify and accelerate the creation of an analysis, we created a tree based editor that represents the attribute tree. This tree is used across all functionalities of the tool. This way, the user avoid entering several times the same information. The tree is created once beforehand and can be reused for multiple analyses. This editor is presented in Figure 4.10.

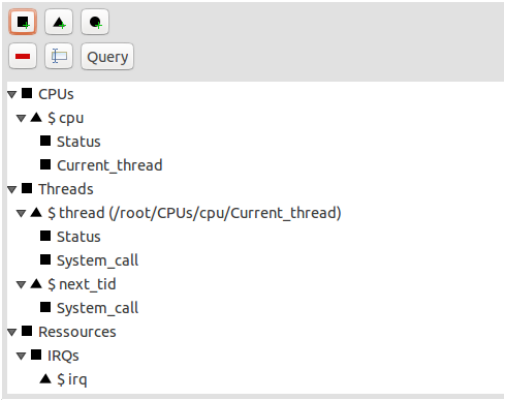

Figure 4.10 The creation of an attribute tree

The Linux kernel analysis is first examined. This use case is a good example and complex enough to show the capabilities of our model, because it covers many concepts and it reimplements a common analysis for the Linux kernel (that is included by default in Trace Compass). Thus, it can easily be compared with the previous implementations. This analysis was initially hard-coded in Java, then represented as a declarative XML specification in Trace Compass, and is now graphically represented in the proposed new tool functionality. It gives us another option to validate and compare our analysis. The first thing to do, to build an analysis, is to build the attribute tree that will be used to specify the state changes and conditions. Figure 4.11 is the tree that will be used to make the state machine. Once this step is done, the state machines can be defined. In this example, we have two state machines. One for CPU status and one for thread status, as mentioned earlier. Figure 4.12 is the complete diagram and Figure 4.13 is an enlarged view of the CPU state machine.

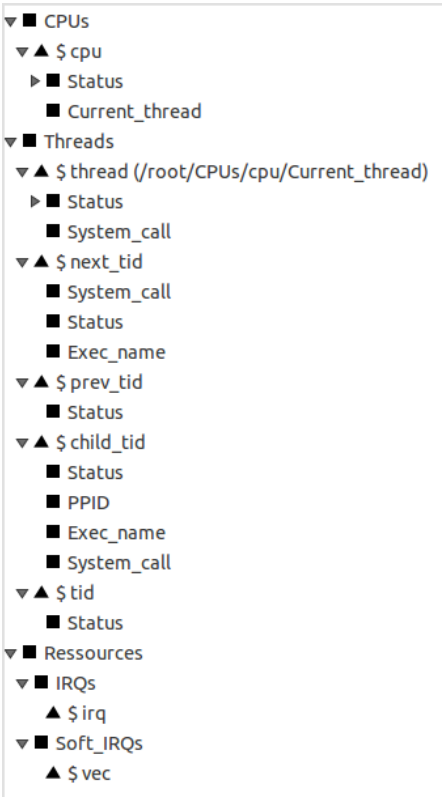

Figure 4.11 Linux kernel attribute tree

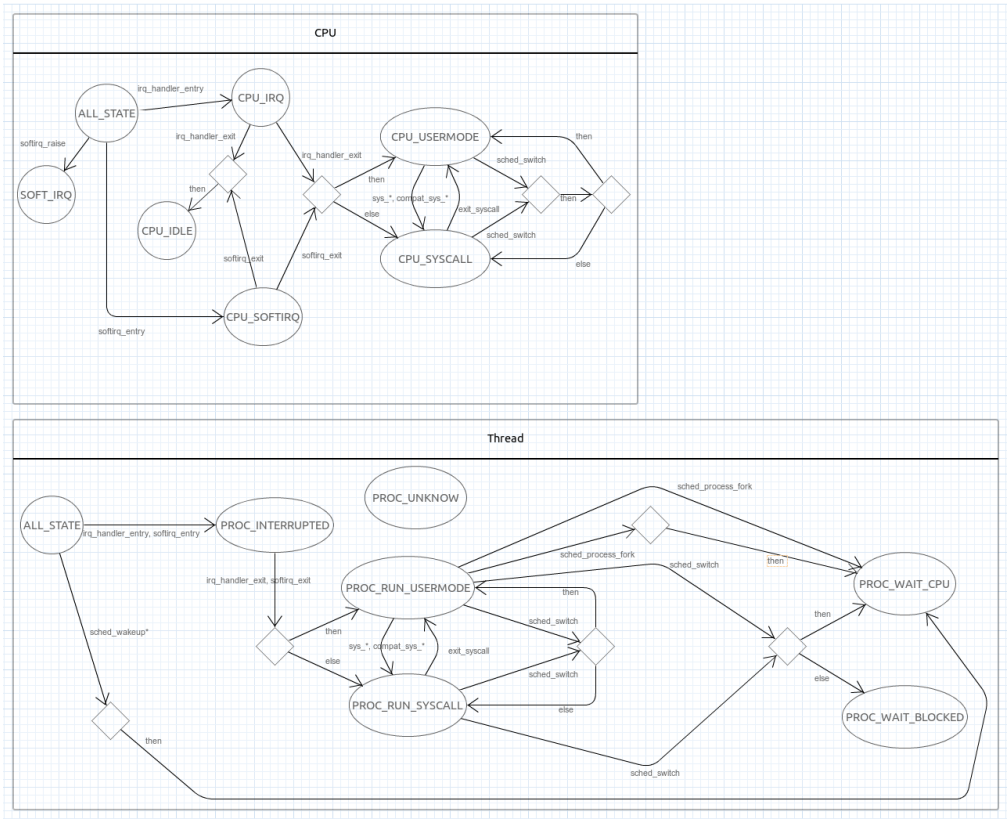

Figure 4.12 State machine of the Linux kernel analysis

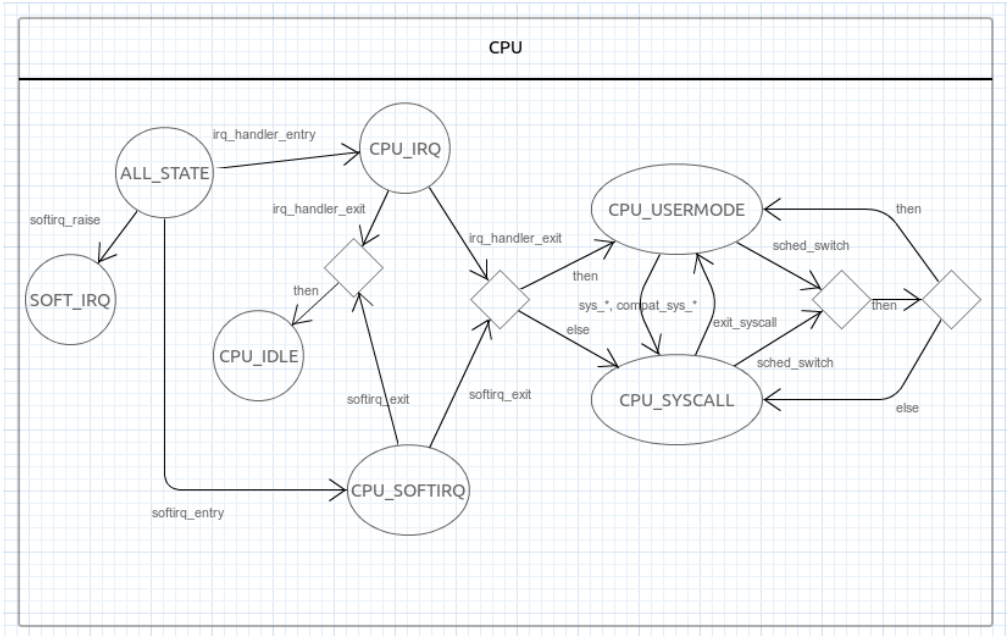

Figure 4.13 CPU state machine used in the Linux kernel analysis

As the tool is intended to be generic, and can be used with any type of trace, a second analysis, for user-space traces, was built to find a particular problem in Trace Compass itself. In some situations, the problem was that some internal requests never ended. To have a better understanding of what is really happening when this problem occurs, a trace is needed. To analyse this problem and find its root cause, we used Trace Compass and our model to extract relevant data on top of that. In the next paragraph, we will show how we used our modeling tool to generate this particular analysis.

Like for the kernel analysis, we need an attribute tree. Figure 4.14 displays this tree and, as can be seen, it is less complex than the kernel analysis tree. Figures 4.15 and 4.16 show the complete analysis. Now more details are provided on this state machine. All the states here are in fact the possible state for a request. In this case, to access these states, the state machine needs to satisfy a condition, represented by the diamond shape. The second figure is an example of how the conditions can be organized. As mentioned, the views are generated from the diagram to be able to see those states when the analysis is completed. First, Figure 4.17 is an example of how to specify the color for a particular state. When everything is specified, we just need to trace the Trace Compass application, capture the problem, and then run the analysis on this trace. The final result is shown in Figure 4.18. With the help of this view, we discovered quickly that requests that never ended were in fact getting into a bad state, something which should not occur.

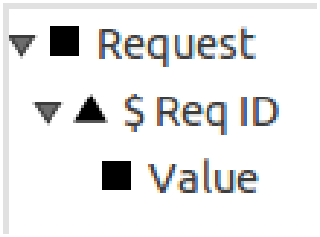

Figure 4.14 Request analysis attribute tree

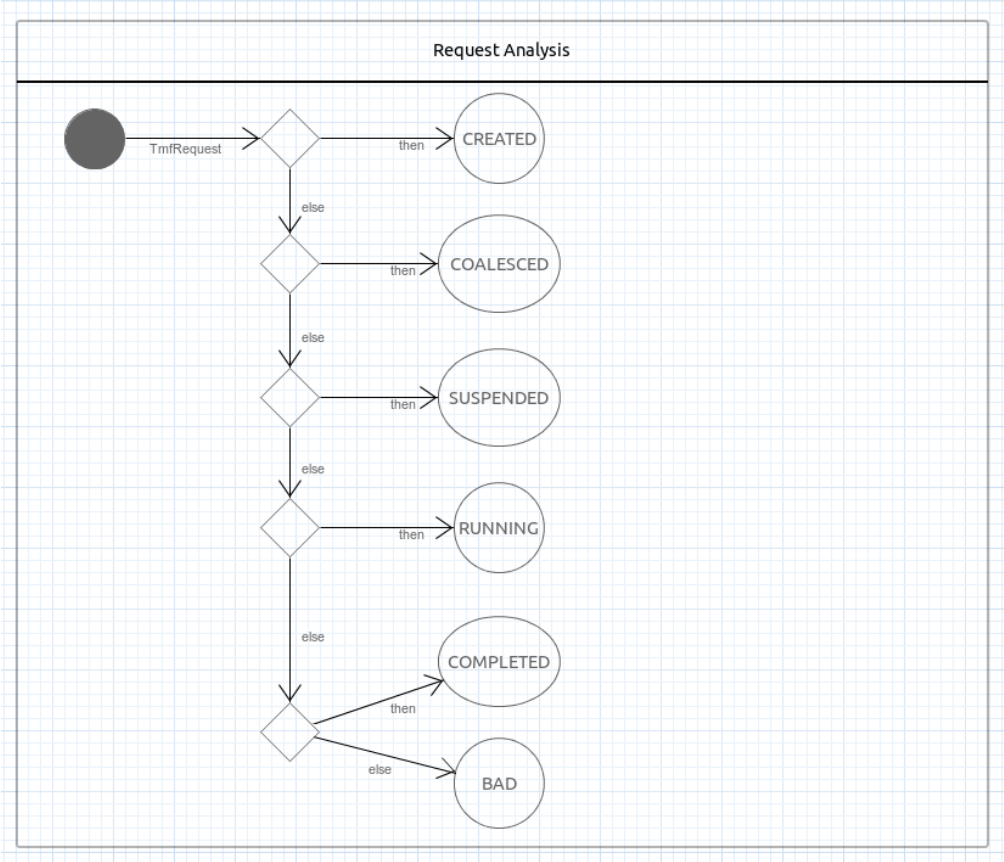

Figure 4.15 State machine of Trace Compass requests analysis

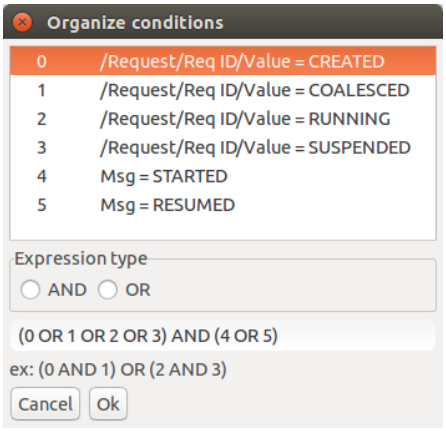

Figure 4.16 Example of one condition used in the requests analysis

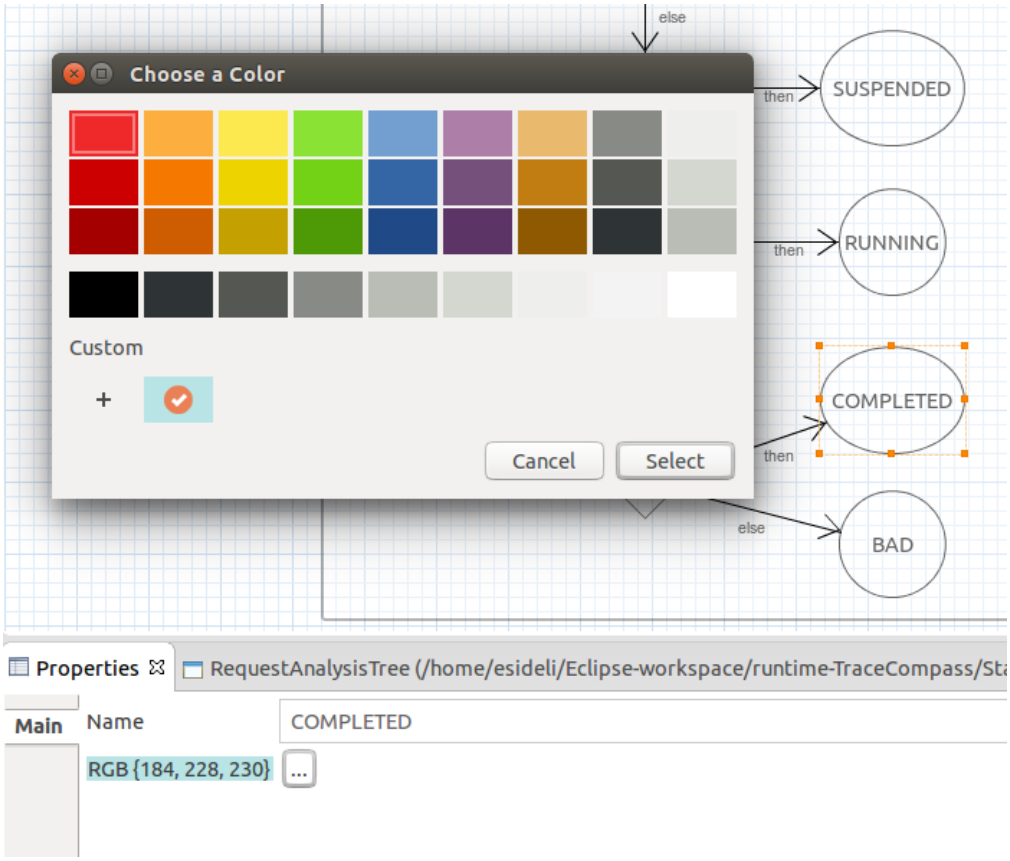

Figure 4.17 Definition of state color for views generation

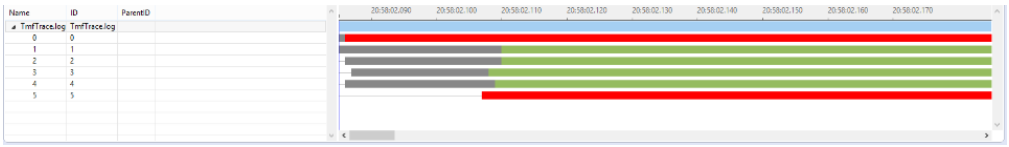

Figure 4.18 View generated from the request analysis

Since these examples were implemented both as XML specifications and graphically in the new proposed modeling tool, this is a good base for comparison. Running these analyses using either XML or a graphical method should give the same result. The advantage of using the graphical method, beside not having to write the XML, is that our model contains more information. With the proposed model, more functionality could be enabled, because of the additional information on transitions and states. We will discuss the possibilities later, in the future work section.

Table 4.1 shows the numbers of attributes for kernel analysis and the requests analysis. In this context, attributes are the properties of an element, for example, the name or the type of a state. In the table, the first column is the old XML specification, the second one is the new model without any diagram information. Note that we did not not account for the file containing the tree for each diagram. These informations are already in the model and the XML. As we can see, there are more attributes in our model. These values are basically all you need to make link between states.

Table 4.1 Number of attributes for different analysis

	XML	Model
Kernel analysis	525	594
Request analysis	196	261

One of our goals is to bring flexibility to the analysis process, by making a generic, powerful and easy to use tool. Since we want our tool to be simple to use, we need to insure that a typical analysis can be accomplished in a short time and will be easy to maintain. For exemple, we were able to model the whole Linux kernel analysis with 14 states, 9 conditions and 36 transitions. Table 4.2 summarizes these numbers. As you can see, the request analysis

Table 4.2 Number of element in the model for different analysis

	Number of states	Number of transitions	Number of conditions
Kernel analysis	14	36	9
Request analysis	7	11	5

is small and contains relatively few elements. This analysis has a single purpose and cannot be reused to solve problem that are not related with the state of a request. To solve this

problem you prefer to not spend a lot of time to write a long analysis that will not be reused often. This is why our tool exists, to make this process simpler and faster.

In summary, the proposed method can easily substitute a significant number of lines of code (ex. Java or XML) and does not need recompiling the analysis tool each time you introduce a new analysis. The modeling tool provides more flexibility to the users and enables them to easily define their own custom analyses based on their problems. Moreover, as mentioned earlier, our tool is generic enough to model different types of problems to analyse.

4.7 Conclusion and Future Work

To find performance issues in applications, tracing combined with a good targeted analysis can be very helpful. Several tools offer interesting analyses. However, for specific and difficult problems, we need custom targeted analyses. This becomes problematic if you then need to access and modify the tool source code to define your own analyses.

In this paper, a modeling tool is proposed to be able to capture all the information related to trace analysis using a state machine. This solution allows the user to define its own analyses and views. The user has the ability to specify the changes that a particular event triggers and how to display each state.

Two examples were detailed to illustrate what can be done with the proposed tool. These two examples cover numerous cases and show how it is possible to use this tool for the various types of traces and applications. Indeed, one example defined a kernel space trace analysis for kernel space and the second a user space trace analysis (Trace Compass custom trace).

There are several possibilities to extend this work. As mentioned earlier, the model contain additional information, as compared to the current XML specification, that can be used to push this tool further. For instance, we could do trace validation, verifying that sequences of events satisfy the specified finite state machines. In the second use case, we could have used this to define the bad state as a validation failure, so when we run the analysis we can check if the trace satisfies the model. The validation will provide hints on which event or series of events caused this failure. The model diagram could be used to display relevant information during the trace analysis. For example, in Trace Compass, when you click on an event, it could highlight the good transition or, if you click on a state in the control flow view, it could show on the diagram which transition caused this change. Other possible improvements include a file/model management system. This tool introduces new files and we need a way to manage these files, whether they come with the trace metadata, from the user preferences, from the tool builtin library or from a group repository. The user should

be able to easily find and apply those analyses to specific traces.

Currently, we use the same state provider as the XML. In fact, we transform our model in an XML file that fits the current implementation. The next step would be to use directly the model, allowing the analysis tool to take advantage of all the future features. Since the model will eventually be used directly, we need a mechanism to validate models. We want to alert the user before he runs the analysis to prevent surprises or unexpected behaviors.

4.8 Acknowledgment

The support of the Natural Sciences and Engineering Research Council of Canada (NSERC), Ericsson Software Research is gratefully acknowledged. The authors would also like to thank Genevieve Bastien and Francis Giraldeau for their help in the open source project Trace Compass.

CHAPITRE 5 DISCUSSION GÉNÉRALE

Dans cette section, nous ferons un bref retour sur les objectifs et les choix effectués, puis nous aborderons différents aspects qui sont en lien avec les résultats obtenus et les travaux futurs discutés précédemment dans le chapitre 4. Nous discuterons entre autres de la façon dont l'intégration des filtres et des patrons pour les événements synthétiques pourrait être faite. De plus, nous aborderons la possibilité d'utiliser le diagramme comme une vue afin de donner aux utilisateurs une autre perspective sur l'analyse par état. Finalement, nous terminerons avec une brève introduction sur la gestion des nouveaux fichiers introduits par notre outil et la façon de rendre l'utilisation des analyses plus conviviale.

5.1 Retour sur les objectifs et la méthodologie

Maintenant que le travail est effectué, nous pouvons revenir sur les objectifs de recherche établis au début et voir s'ils ont été atteints. Tout d'abord, nous avons réussi à créer un modèle qui supporte toutes les fonctionnalités de l'ancienne méthode. Notre outil permet donc de remplacer totalement l'écriture du fichier XML par l'utilisateur. De plus, notre éditeur d'arbre d'attributs permet de faciliter la construction d'analyses en évitant de saisir à plusieurs reprises la même information dans la machine à état. À la suite des résultats obtenus, nous pouvons dire qu'il est possible de spécifier l'état d'un système grâce à notre outil et ainsi générer une analyse qui donne le même résultat qu'une analyse déclarative faite en XML. Nous avons conçu l'analyse du noyau Linux, car elle contient toutes les fonctionnalités les plus avancées possible dans le XML. De cette façon, il est possible de dire si notre outil peut remplacer l'ancienne méthode, ce qui a été confirmé. Les différentes parties d'une analyse XML sont la définition du fournisseur d'état et la définition des vues. À l'intérieur du fournisseur, il y a la spécification des changements d'état et les changements d'état conditionnel. Dans notre solution, toutes ces parties ont été couvertes. Le fournisseur d'état est le diagramme lui-même, c'est-à-dire, la ou les machines à état que l'utilisateur construira constitue ce fournisseur. Les changements d'état sont couverts par les transitions et les informations qu'elles renferment. Finalement, les changements d'état conditionnels sont couverts par les pseudo états `Condition`. Ils contiennent l'information sur la ou les conditions à vérifier. Les transitions sortantes sont les différents changements d'état possibles.

Dans la méthodologie (chapitre 3), nous avons énoncé les critères de sélection pour le cadriciel graphique que nous allions utiliser. Après plusieurs prototypes et essais de ces différents cadriciels, nous en sommes arrivés à utiliser Graphiti au lieu de GMF ou Sirius. En pre-

mier lieu, nous avons éliminé GMF, car le développement n'est plus très actif. Comme le développement, la communauté n'est plus très active non plus. Nous avons vérifié la vitalité de la communauté en posant quelques questions techniques sur les forums afin de nous aider à développer notre outil. Malheureusement, elles sont restées sans réponse. De plus, le développement est fastidieux et de nombreux bogues sont présents. En ce qui concerne Graphiti et Sirius, les deux possèdent une communauté active et les réponses aux questions furent rapides. L'utilisation de ces cadriciels est facile et permet de faire des prototypes très rapidement et ainsi voir les possibilités offertes. L'interface graphique est semblable dans les deux cas. Le point qui a fait la différence dans notre décision est l'intégration possible à Trace Compass. En effet, Sirius utilise une interface graphique pour créer l'outil, donc aucune programmation n'est nécessaire. Par contre, il est plus difficile de l'intégrer au sein d'un autre outil. Pour Graphiti, une interface de programmation est offerte et elle est facile d'utilisation. Les prototypes initiaux ont vite montré qu'il était facile de l'intégrer comme un module d'extension pour Trace Compass.

Bref, notre outil permet de couvrir toutes les fonctionnalités offertes précédemment avec le XML, et ce en utilisant seulement une interface graphique. Pour y arriver, nous avons utilisé Graphiti pour faire l'éditeur de diagrammes, puis nous l'avons intégré dans Trace Compass.

5.2 Intégration de nouvelle fonctionnalités

Puisque la modélisation dans le domaine du traçage est très jeune, les possibilités d'amélioration sont grandes. L'introduction de notre outil dans Trace Compass ouvre la porte à l'expansion de ce type de fonctionnalité dans les logiciels d'analyse de traces. Éventuellement, toutes les étapes du traçage pourront se faire à l'aide de modélisation.

Au moment de nos travaux, deux fonctionnalités retiennent notre attention. Il y a la spécification de filtres et la génération d'événements synthétiques. Ceux-ci peuvent être modélisés sans trop de modifications au modèle utilisé dans ce mémoire. Par contre, quelques questions de convivialité devront être résolues afin de rendre l'outil efficace avec l'ajout de ces deux nouvelles fonctionnalités.

Tout d'abord, nous allons expliquer brièvement ces travaux afin de bien saisir le contexte. Puis, nous énoncerons quelques pistes de solution possibles.

5.2.1 Filtres

Comme une trace d'exécution peut contenir beaucoup d'événements, le besoin de filtrer ceux-ci est bien présent. En effet, les développeurs ne veulent pas nécessairement avoir tous les

événements. Ils ont donc besoin d'un moyen pour les filtrer.

Un collègue dans le laboratoire DORSAL a proposé une solution de filtrage d'événements guidé par les données. Un filtre est défini par une machine à états finis sauvegardée dans un fichier XML. Pour le moment, le fichier est écrit à la main. Pour cette raison, la modélisation graphique permettrait à l'utilisateur d'être plus efficace et de mieux comprendre les filtres définis.

Présentement, les filtres sont définis à l'aide de trois composantes principales :

— L'événement ou un groupe d'événements d'entrée, qui marquent le début du filtre.
— Définition des actions possibles lors d'une transition.
— La machine à état. C'est-à-dire les états et les transitions entre ceux-ci.

5.2.2 Événement synthétique

Comme les traces fournissent des événements spécifiques au système tracé, il faut adapter l'outil d'analyse pour supporter ces différents événements. Une solution présentée par Ezzati-Jivan and Dagenais (2012) permet de créer des événements dits synthétiques. Ces événements représentent un ensemble d'événements tirés de la trace qui, mis ensemble, décrivent un comportement du système. La lecture séquentielle d'un fichier est un exemple d'événement synthétique.

Une approche par état a été utilisée pour bâtir le modèle d'un événement synthétique. C'est-à-dire que nous utilisons une machine à états comme modèle pour déterminer si une suite d'états représente un événement synthétique. Les machines à état représentent une série d'actions et de transitions qui mèneront à un événement synthétique. Ce problème s'applique bien à la modélisation, en particulier avec une machine à état. En effet, cette méthode fait abstraction du système tracé et définit des comportements. En UML les machines à état sont faites pour représenter un comportement.

5.2.3 Ajout à l'outil de modélisation

Maintenant que nous avons une idée des possibilités de fonctionnalité qui peuvent être utilisées avec de la modélisation, nous aborderons les différentes solutions d'intégration dans notre outil.

Tout d'abord, ces deux solutions sont basées sur un modèle de machine à état et reprennent le concept de changement d'état (*state change*). C'est pourquoi l'intégration est possible et requiert peu de modifications au modèle conçu dans ce mémoire. Avant d'exposer les solutions, il faut connaître les raisons pour lesquelles nous voulons intégrer ces fonctionnalités à même

l'outil. La raison principale est que cela offre plus de possibilités à l'utilisateur, directement intégré dans l'outil d'analyse de traces, afin de créer des analyses plus complètes et plus adaptées à leurs problèmes.

L'intégration de ces fonctionnalités ne doit pas être faite de façon isolée et il faut les voir comme un tout qui permet de complémenter les autres fonctionnalités. En effet, chaque machine à état doit fournir de l'information supplémentaire qui peut être utilisée par les autres analyses. Les trois fonctionnalités discutées ici sont : les filtres, les événements synthétiques et notre contributions concernant les analyses à l'aide d'une machine à état.

Dans le cas des filtres, les événements créés lors de la conception de l'analyse de la trace peuvent être réutilisés afin de minimiser le temps de création de ceux-ci. Non seulement les événements peuvent être réutilisés, mais les transitions, les états et même des parties de machine à état peuvent l'être aussi. En ce qui concerne les filtres, l'analyse à l'aide de la machine à état ne bénéficie pas beaucoup de cet ajout, mais davantage l'inverse.

Les événements synthétiques peuvent bénéficier de la machine à état provenant de l'analyse, de la même façon que les filtres, c'est-à-dire en réutilisant les éléments déjà créés. Par contre, comme cette technique permet de créer en quelque sorte des événements, elle peut donc contribuer à améliorer la construction de l'analyse. En effet, les événements synthétiques pourraient être utilisés comme l'événement qui déclenchera une transition dans l'analyse ou dans les filtres. Donc, les changements d'état dans une analyse peuvent être représentés par des comportements plutôt que des événements ponctuels.

Maintenant que les liens entre les fonctionnalités sont faits, il faut que le tout s'intègre bien dans l'outil et que l'utilisateur puisse facilement atteindre le but voulu. L'idée principale est d'intégrer les trois diagrammes dans la même vue. Ceci implique aussi d'intégrer les trois modèles dans un même fichier, afin de ne pas avoir de problèmes de dépendances et de fichiers inexistants qui pourraient corrompre les différents modèles. La solution proposée est de réutiliser le concept utilisé dans Papyrus. Il affiche plusieurs diagrammes en utilisant des onglets. La Figure 5.1 montre un exemple de la façon dont on peut représenter plusieurs diagrammes dans une même vue. C'est aussi une façon d'indiquer que ces diagrammes sont en relation et qu'ils peuvent faire des liens entre les différents éléments de la machine à état.

Bref, toutes ces fonctionnalités peuvent coexister au sein d'un même outil afin de minimiser le travail en double et permettent à l'utilisateur de concevoir des analyses plus complexes et précises.

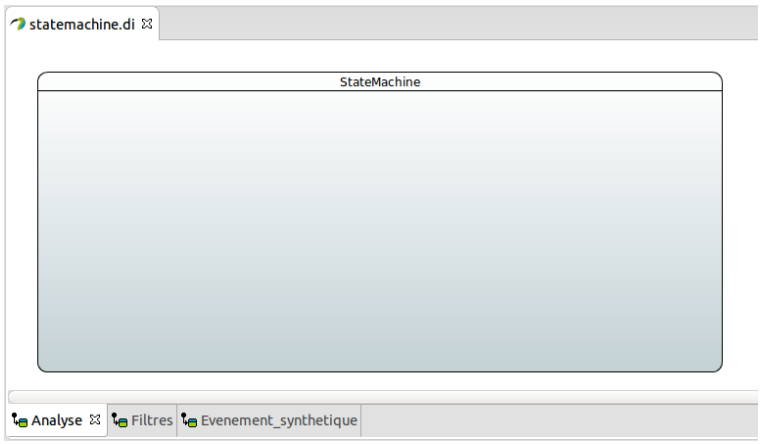

Figure 5.1 Exemple de vue utilisée dans Papyrus

5.3 Utilisation de l'outil comme une vue

Maintenant, abordons quelques fonctionnalités futures de mon outil, plus précisément les possibilités de l'utiliser comme une vue dans un logiciel d'analyse. Comme la machine à état représente l'analyse elle-même, il est possible d'y afficher de l'information résultant de cette analyse.

Premièrement, nous pourrions synchroniser cette vue avec les autres vues de Trace Compass et ainsi la rendre plus dynamique. En effet, nous pourrions mettre en surbrillance les états et les transitions qui ont mené à celui-ci, par exemple, lors d'un clic dans la vue de flux de commande (*control flow view*). De plus, nous pourrions tenter de rejouer une séquence de la trace afin de voir par quel état un certain attribut passe.

Finalement, nous pourrions rendre cette vue interactive afin de synchroniser les autres vues en fonction des entrées de l'utilisateur sur la machine à état. En effet, nous pourrions créer des filtres rapides, différents des filtres mentionnés plus haut, en utilisant les transitions. Ces filtres pourraient servir à montrer ce type d'événement seulement dans la table d'événements. De plus, les états pourraient contenir de l'information tirée de l'analyse. Par exemple, l'utilisateur pourrait naviguer dans l'arbre pour voir les valeurs des différents attributs dans cet état.

5.4 Gestion des fichiers et préférences

Afin de rendre l'outil d'analyse accessible à tous les utilisateurs, il faut faciliter son utilisation en s'assurant d'avoir une bonne cohésion et d'éliminer les étapes ou la gestion inutile. Étant donné que notre outil introduit différents fichiers et que dans le futur plusieurs fichiers et fonc-

tionnalités s'ajouteront, il faut une façon conviviale de gérer le tout. Cette gestion n'affecte pas seulement la modélisation. Les préférences et autres fonctionnalités de Trace Compass peuvent en bénéficier. La présente section discute brièvement d'organisations possibles pour ces fichiers et préférences.

5.4.1 Nouveaux fichiers

Tout d'abord, exposons les nouveaux fichiers et leur rôle dans Trace Compass. Les trois principaux ajouts sont le diagramme, l'arbre d'attributs et l'analyse qui combine les deux fichiers précédents.

Le diagramme contient toutes les informations contenues dans le modèle ainsi que des éléments graphiques propres à Graphiti. L'arbre d'attributs contient la structure de l'arbre. Ce fichier ne contient aucune valeur propre à l'analyse afin de pouvoir le réutiliser dans d'autres contextes. Finalement, l'analyse qui en découle de ces deux fichiers sera utilisée pour exécuter l'analyse. Il contient l'information du modèle sans les informations graphiques.

5.4.2 Interaction entre les fichiers

Avant de donner une solution pour la gestion de ces fichiers, nous devons comprendre toutes les dépendances qui peuvent exister entre eux. Puisque l'arbre d'attributs contribue en grande partie à la machine à état, le diagramme dépend de cet arbre. Les filtres et les événements synthétiques dépendent aussi de celui-ci. En effet, le diagramme doit utiliser toujours le même arbre afin d'éviter les références invalides. Par contre, l'arbre ne dépend pas des autres composantes. Le diagramme peut avoir un lien avec les événements synthétiques. L'analyse est générée à partir du diagramme et peut contenir possiblement plusieurs diagrammes. Ces diagrammes peuvent représenter une autre analyse pour la même trace, les filtres ou encore la spécification d'événements synthétiques.

La Figure 5.2 résume bien les dépendances. Nous voyons que la trace est la racine dans ce cas, mais elle pourrait être une expérimentation dans Trace Compass. Ensuite, l'analyse possède un diagramme qui est lié avec un arbre d'attributs. Finalement, les filtres et les événements synthétiques sont présents sous la trace.

5.4.3 Architecture possible

Comme mentionné précédemment, nous devons avoir une bonne gestion de tous ces fichiers ainsi que d'autres éléments de Trace Compass comme les préférences. Avoir une façon de gérer tous ces éléments à un seul endroit dans l'outil permet de faciliter la tâche et la compréhension

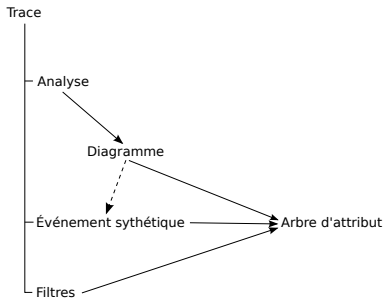

Figure 5.2 Dépendances entres les différents fichiers

pour l'utilisateur. Présentement, la configuration et la gestion des éléments d'une analyse se font de façon individuelle et dispersée dans l'outil d'analyse. Il faut donc avoir une gestion globale qui réunit tous ces éléments à un seul endroit. De plus, il sera plus facile d'accéder aux valeurs courantes et de les modifier. Il serait aussi possible d'exporter ou d'importer les préférences utilisateur.

La piste de solution que nous proposons est basée sur des systèmes de configuration utilisés par GNOME[1] sur les systèmes Linux. Précédemment, *gconf* était le gestionnaire pour les configurations et les préférences utilisateur. Il utilisait des fichiers XML afin de spécifier les schémas. Ces derniers sont utilisés pour organiser les clés et elles sont organisées un peu comme des dossiers. Par exemple, nous pouvons avoir /app/... ou /system/... pour les préférences par application ou pour le système. L'architecture de *gconf* proposait trois fichiers : un pour les configurations obligatoires qui avait la priorité sur les autres fichiers, un pour les préférences et configurations de l'utilisateur qui est traité après les préférences obligatoires et un contenant les valeurs par défaut et est traité à la fin. Par la suite, *gconf* a été remplacé par *dconf* et *gsettings*. La raison de ce changement fut la performance. En effet, les fichiers XML sont peu efficaces en lecture et un fichier de configuration a beaucoup d'accès en lecture lors du démarrage du système ou de l'application. C'est la raison pour laquelle la solution de rechange utilise des fichiers binaires optimisés pour la lecture. *dconf* et *gsettings* fonctionnent ensemble. C'est-à-dire que *gsettings* est une interface de programmation pour obtenir, ajouter ou modifier les valeurs contenues dans la base de données *dconf*. Ce dernier ressemble à son prédécesseur en termes de schéma et d'organisation des préférences. Les schémas sont définis avec les champs suivants : un identificateur, un chemin d'accès, une clé, une description et un résumé, puis une valeur par défaut. *Dconf* n'utilise pas le même système que *gconf* pour la hiérarchie des configurations. Habituellement, la base de données de l'utilisateur a priorité sur les autres, puis celle du système. Afin de permettre des clés

1. `https://www.gnome.org/`

obligatoires, il existe un système de verrouillage. Ce système permet de verrouiller une clé et les bases de données au-dessus dans la hiérarchie ne pourront pas la modifier. Par exemple, si une clé est verrouillée dans le système, alors l'utilisateur ne pourra pas la modifier.

En utilisant ces principes, nous pouvons concevoir une architecture qui comblerait nos besoins. Nous devons concevoir un gestionnaire de configuration qui réunit les différents niveaux présents dans Trace Compass. En effet, il y a des paramètres pour Trace Compass lui-même, pour les traces et pour les expérimentations. Puis, nous pourrions avoir des préférences par trace ou par type de trace. La Figure 5.3 permet de visualiser les différents niveaux possibles dans Trace Compass.

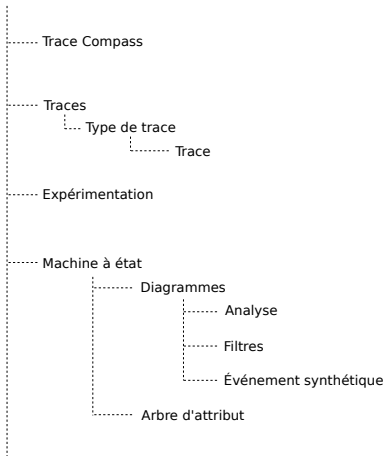

Figure 5.3 Exemple de hiérarchie pour les préférences de Trace Compass

En s'inspirant des solutions déjà proposées dans Linux, nous pouvons appliquer une solution similaire à *dconf* en termes de structure des préférences et du concept de verrouillage. Il faut aussi respecter une certaine hiérarchie lors de la lecture des fichiers de configuration. C'est pourquoi nous avons pensé utiliser une structure en arbre un peu comme un système de fichiers et des chemins d'accès comme *dconf*. En utilisant ce type de structure, il est possible d'établir un certain ordre d'exécution. Dans notre cas, nous voulons que les préférences des noeuds remplacent les préférences de leurs parents, dans le cas où les clés sont les mêmes. Par exemple, l'analyse à utiliser lorsqu'une trace est ouverte serait définie dans la clé `Trace`. Par contre, si l'utilisateur définit qu'une analyse ne s'applique qu'à un seul type de trace, il ajoutera cette entrée dans la clé `Type de trace`. Dans ce cas, l'analyse à utiliser dans `Type de trace` doit avoir préséance sur la clé `Trace`. Chaque noeud de l'arbre peut avoir plusieurs entrées. Par exemple, la clé Analyse peut avoir une entrée pour l'arbre d'attributs associé et une autre la taille des caractères dans l'éditeur de la machine à état. Un autre exemple de clé est que la `Trace` peut avoir la machine à état d'une analyse comme entrée.

Voici quelques exemples de clés :

```
/FSA/Diagrammes -> font_size = 12
/FSA/Diagrammes/AnalyseA -> tree = /chemin/de/l'arbre
/FSA/Diagrammes/AnalyseB -> tree = /chemin/d'un/autre/arbre, font_size = 16
```

Afin de permettre les configurations obligatoires pour les utilisateurs, nous reprenons le concept de verrouillage de clé. Cela consiste à empêcher la surcharge d'une clé par un enfant. Par exemple, si la clé `Diagramme` spécifie que la taille du texte est de 12 points et que la clé est verrouillée, alors il est impossible d'utiliser différentes tailles de texte pour l'analyse et pour les filtres. La clé `/FSA/Diagrammes/AnalyseB -> font_size = 16` devient inutile et sera ignorée.

En résumé, nous avons une structure en arbre qui permet de conserver une certaine hiérarchie dans les préférences et configurations. De cette façon, les utilisateurs peuvent surcharger certaines préférences pour certaines clés. Il peut y avoir des préférences par trace, par type de trace ou pour une expérimentation. Par exemple, dans la figure précédente, Trace Compass peut avoir des préférences plus générales, comme la langue ou le fuseau horaire. Plus on descend dans l'arbre plus on est susceptible de rencontrer des paramètres plus spécifiques comme l'analyse à utiliser pour un type de trace. Par contre, rien n'empêche de changer le fuseau horaire pour une trace en particulier, afin que l'analyse et que l'affichage restent cohérents. Cette solution permet de regrouper tous les fichiers de configuration et de préférences à un seul endroit et ainsi faciliter la gestion. Il devient alors plus facile pour un utilisateur de comprendre le comportement général de l'application.

CHAPITRE 6 CONCLUSION ET RECOMMANDATIONS

6.1 Synthèse des travaux

Quand vient le temps d'identifier des problèmes de performance, le traçage, combiné à de bonnes analyses ciblées, peut s'avérer utile. Plusieurs outils sont disponibles et ils offrent souvent plusieurs analyses prêtes à être utilisées. Par contre, les besoins sont différents en fonction du contexte et du but de notre application. Souvent, ces analyses sont génériques et parfois nous avons besoin d'analyses plus précises.

La problématique est qu'il faut avoir accès au code source de l'outil afin d'y ajouter des analyses personnalisées. Depuis peu, il est possible d'utiliser une méthode dirigée par les données pour spécifier des analyses. Cette méthode remplace le fournisseur d'état écrit en Java par un langage déclaratif écrit dans un fichier XML. L'inconvénient de cette solution est l'écriture du fichier lui-même. En effet, il peut rapidement devenir volumineux et ainsi introduire des erreurs.

Pour remédier à ces problèmes, nous avons présenté un outil de modélisation pour capturer toutes les informations reliées à l'analyse de traces, et ce à l'aide d'un éditeur de machine à état. Cet outil permet à l'utilisateur de bâtir sa propre analyse ainsi que de spécifier des vues. L'utilisateur a maintenant le pouvoir de définir les changements d'états qu'un événement provoque et l'affichage de ceux-ci dans le logiciel d'analyse.

Nous avons présenté un modèle de machine à état qui est adapté au domaine du traçage. Ce modèle est inspiré du UML afin de permettre un apprentissage plus rapide pour les utilisateurs familiers avec la modélisation UML. L'implémentation a été faite en utilisant EMF. Pour manipuler le modèle, nous avons utilisé le cadriciel Graphiti, car il offrait plus de flexibilité pour personnaliser l'affichage, une intégration facile à Trace Compass et une communauté active. En combinant ces deux technologies, nous sommes arrivés à concevoir un outil complet et facile d'utilisation.

Afin de valider l'exactitude de notre solution, nous avons montré quelques exemples d'utilisation. Ces exemples couvrent plusieurs cas d'utilisation et montrent l'application de l'outil avec différents types de traces (trace noyau et trace propre à Trace Compass) et différentes applications (espace noyau et espace utilisateur).

De plus, cette solution permet d'y intégrer d'autres fonctionnalités qui permettront de rendre l'outil d'analyse encore plus efficace. Par exemple, l'utilisation du diagramme comme vue, l'intégration des filtres et des événements synthétiques et la validation de traces sont possibles

dans cette architecture.

Les résultats présentés dans l'article montrent qu'il est possible de modéliser une analyse à l'aide d'un éditeur de machine à état fini.

6.2 Limitations de la solution proposée

Tout comme la spécification du fournisseur d'état en XML présentement, notre solution se veut un remplacement de celui-ci. Nous sommes donc limités dans les performances d'exécution de nos analyses. En effet, la construction du gestionnaire d'état se fait de façon séquentielle et il n'est pas possible de commencer l'analyse à n'importe quel point de la trace, car la méthode actuelle prend pour acquis que les événements arrivent en ordre chronologique.

Une autre limitation est que le gestionnaire d'états ne contient aucune information sur les dépendances entre les différents états. Notre modèle contient ces informations, mais il faut modifier le gestionnaire pour prendre en compte ces informations sur la provenance des transitions. Par contre, cette modification apporte son lot de défis, car il faut conserver la performance de la construction et de la recherche dans le gestionnaire d'états .

Présentement, notre modèle n'est pas directement utilisé dans Trace Compass. En effet, les informations du modèle sont extraites, puis transformées pour générer le XML. Cette transformation permet d'utiliser les mécanismes déjà présents dans le logiciel d'analyse. Par contre, nous sommes limités par les fonctionnalités du XML. Il faudrait utiliser le modèle directement afin de profiter pleinement de ses fonctionnalités.

6.3 Améliorations futures

Comme la modélisation est relativement récente dans le domaine du traçage, les possibilités sont grandes. Plus précisément, la construction d'analyses peut être améliorée avec différentes fonctionnalités. Dans le chapitre chapitre 5 nous avons discuté de quelques pistes de solution pour les travaux futurs. Nous avons abordé la possibilité d'utiliser l'outil de modélisation comme une vue, afin de rendre les machines à état interactives et ainsi permettre de naviguer dans la trace . De plus, l'intégration des filtres et des événements synthétiques ajouterait deux fonctionnalités qui pourraient grandement bénéficier aux analyses conçues dans notre outil. En effet, les relations entre ces différentes fonctionnalités permettent de concevoir des analyses plus complexes.

Puisque l'outil ajoute un autre niveau de préférence et de configuration à l'outil d'analyse actuel, il serait intéressant d'avoir une bonne gestion de ces paramètres. Cela permettrait

d'avoir des préférences par trace ou par type de trace par exemple.

Une amélioration qui n'a pas été discutée encore est la possibilité d'utiliser le modèle créé dans ce mémoire afin de valider une trace. Par exemple, si l'on bâtit une analyse qui comporte un état `corrompu` qui ne devrait pas être atteint lors d'une exécution normale du programme, on pourrait avertir le développeur de la série d'événements qui a mené à cet état.

Comme mentionné dans les limitations, il faudrait changer la façon dont le gestionnaire de système est construit. En effet, il faudrait modifier le fournisseur d'état pour utiliser directement notre modèle et éventuellement changer le système d'état actuel pour y ajouter plus d'information, que notre modèle possède déjà, et qui sont manquantes pour faire les liens entre les différents états, comme déterminer la source de la transition.

Nous avons mentionné brièvement qu'il était possible de générer des vues à partir de notre modèle. La vue générée est de type diagramme de Gantt, mais il est maintenant possible de définir des graphiques de type xy. Par contre, nous ne supportons pas encore la génération de ce type de vue. Il serait donc intéressant que l'utilisateur puisse les définir avec une interface graphique qui utilise les entrées qu'il a faites dans le modèle de machine à état.

Afin de rendre l'outil plus robuste, il serait intéressant d'introduire une vérification du modèle. De cette façon, nous pourrions mieux guider l'utilisateur si le modèle ne respecte pas les différentes contraintes.

Dans un futur plus éloigné, les applications de la modélisation pourraient être plus présentes dans le domaine de l'analyse de performance. Il serait intéressant d'intégrer le traçage et l'analyse de traces dans les outils de conception logiciel déjà utilisés depuis plusieurs années (ex. Papyrus). Il serait alors possible de spécifier les points de trace et de construire des analyses en utilisant les diagrammes qui servent à générer le code de l'application. L'accessibilité aux outils de traçage en sera grandement améliorée.

RÉFÉRENCES

T. Beauchamp et D. Weston, “Dtrace : The reverse engineer’s unexpected swiss army knife”, *Blackhat Europe*, 2008.

C. Brand, M. Gorning, T. Kaiser, J. Pasch, et M. Wenz, “Development of high-quality graphical model editors”, *Eclipse Magazine*, 2011.

B. Cantrill, M. W. Shapiro, A. H. Leventhal *et al.*, “Dynamic instrumentation of production systems.” dans *USENIX Annual Technical Conference, General Track*, 2004, pp. 15–28.

A. Chan, W. Gropp, et E. Lusk, “An efficient format for nearly constant-time access to arbitrary time intervals in large trace files”, *Scientific Programming*, vol. 16, no. 2, pp. 155–165, 2008.

Corbet. (2007) Introducing utrace. En ligne : `https://lwn.net/Articles/224772/`

M. Desnoyers. (2011) Common trace format (ctf). En ligne : `http://www.efficios.com/ctf`

M. Desnoyers et M. R. Dagenais, “The lttng tracer : A low impact performance and behavior monitor for gnu/linux”, dans *OLS (Ottawa Linux Symposium)*, vol. 2006. Citeseer, 2006, pp. 209–224.

S. T. Eckmann, G. Vigna, et R. A. Kemmerer, “Statl : An attack language for state-based intrusion detection”, *Journal of computer security*, vol. 10, no. 1, pp. 71–103, 2002.

J. Edge. (2009) Perfcounters added to the mainline. En ligne : `http://lwn.net/Articles/339361/`

F. C. Eigler et R. Hat, “Problem solving with systemtap”, dans *Proc. of the Ottawa Linux Symposium*. Citeseer, 2006, pp. 261–268.

N. Ezzati-Jivan et M. R. Dagenais, “A stateful approach to generate synthetic events from kernel traces”, *Advances in Software Engineering*, vol. 2012, p. 6, 2012.

——, “A framework to compute statistics of system parameters from very large trace files”, *ACM SIGOPS Operating Systems Review*, vol. 47, no. 1, pp. 43–54, 2013.

——, “Cube data model for multilevel statistics computation of live execution traces”, *Concurrency and Computation : Practice and Experience*, 2014.

P.-M. Fournier, M. Desnoyers, et M. R. Dagenais, “Combined tracing of the kernel and applications with lttng”, dans *Proceedings of the 2009 linux symposium*, 2009.

S. Gérard, C. Dumoulin, P. Tessier, et B. Selic, “19 papyrus : A uml2 tool for domain-specific language modeling”, dans *Model-Based Engineering of Embedded Real-Time Systems*. Springer, 2010, pp. 361–368.

F. Giraldeau, J. Desfossez, D. Goulet, M. Dagenais, et M. Desnoyers, “Recovering system metrics from kernel trace”, dans *Linux Symposium*, vol. 109, 2011.

S. Goswami. (2005) An introduction to kprobes. En ligne : `https://lwn.net/Articles/132196/`

B. Gregg et J. Mauro, *DTrace : Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD*. Prentice Hall Professional, 2011.

R. B. Insung Park. (2007) Improve debugging and performance tuning with etw. En ligne : `https://msdn.microsoft.com/en-us/magazine/cc163437.aspx`

S. D. Jim Keniston. (2010) Uprobes : User-space probes. En ligne : `http://events.linuxfoundation.org/slides/lfcs2010_keniston.pdf`

K. Kouame, N. Ezzati-Jivan, et M. R. Dagenais, “A flexible data-driven approach for execution trace filtering”, dans *Big Data (BigData Congress), 2015 IEEE International Congress on*. IEEE, 2015, pp. 698–703.

D. G. Mathieu Desnoyers, Julien Desfossez. (2012) Lttng 2.0 : Tracing for power users and developers. En ligne : `http://lwn.net/Articles/491510/`

A. Montplaisir, N. Ezzati-Jivan, F. Wininger, et M. Dagenais, “Efficient model to query and visualize the system states extracted from trace data”, dans *Runtime Verification*. Springer, 2013, pp. 219–234.

A. Montplaisir-Gonçalves, N. Ezzati-Jivan, F. Wininger, et M. R. Dagenais, “State history tree : an incremental disk-based data structure for very large interval data”, dans *Social Computing (SocialCom), 2013 International Conference on*. IEEE, 2013, pp. 716–724.

Object Management Group (OMG), “Omg unified modeling language™”, Sep. 2013. En ligne : http://www.omg.org/spec/UML/2.5/Beta2/PDF

V. Prasad, W. Cohen, F. Eigler, M. Hunt, J. Keniston, et J. Chen, “Locating system problems using dynamic instrumentation”, dans *2005 Ottawa Linux Symposium*. Citeseer, 2005, pp. 49–64.

M. Roesch *et al.*, “Snort : Lightweight intrusion detection for networks.” dans *LISA*, vol. 99, no. 1, 1999, pp. 229–238.

S. Rostedt. (2010) Using the trace_event() macro. En ligne : http://lwn.net/Articles/379903/

D. Steinberg, F. Budinsky, E. Merks, et M. Paternostro, *EMF : eclipse modeling framework*. Pearson Education, 2008.

C. Stritzke et S. Lehrig, “Why and how we should use graphiti to implement pcm”, 2013.

A. Van Deursen, P. Klint, et J. Visser, “Domain-specific languages : An annotated bibliography.” *Sigplan Notices*, vol. 35, no. 6, pp. 26–36, 2000.

V. Viyovic, M. Maksimovic, et B. Perisic, “Sirius : A rapid development of dsm graphical editor”, dans *Intelligent Engineering Systems (INES), 2014 18th International Conference on*. IEEE, 2014, pp. 233–238.

V. Vujović, M. Maksimović, et B. Perišić, “Comparative analysis of dsm graphical editor frameworks : Graphiti vs. sirius”.

F. Wininger, “Conception flexible d’analyses issues d’une trace système”, Mémoire de maîtrise, École Polytechnique de Montréal, 2014.

O. Zaki, E. Lusk, W. Gropp, et D. Swider, “Toward scalable performance visualization with jumpshot”, *International Journal of High Performance Computing Applications*, vol. 13, no. 3, pp. 277–288, 1999.